3-14-2014

Online Build-Order Optimization for Real-Time Strategy Agents Using Multi-Objective Evolutionary Algorithms

Jason M. Blackford

ONLINE BUILD-ORDER OPTIMIZATION FOR REAL-TIME STRATEGY AGENTS USING MULTI-OBJECTIVE EVOLUTIONARY ALGORITHMS

THESIS

Jason M. Blackford, Captain, USAF

AFIT-ENG-14-M-13

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-14-M-13

ONLINE BUILD-ORDER OPTIMIZATION FOR REAL-TIME STRATEGY AGENTS USING MULTI-OBJECTIVE EVOLUTIONARY ALGORITHMS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Jason M. Blackford, B.S.C.p.E.

Captain, USAF

March 2014

AFIT-ENG-14-M-13

ONLINE BUILD-ORDER OPTIMIZATION FOR REAL-TIME STRATEGY AGENTS USING MULTI-OBJECTIVE EVOLUTIONARY ALGORITHMS

Jason M. Blackford, B.S.C.p.E.
Captain, USAF

Approved:

//signed//	14 Mar 2014
Gary B. Lamont, PhD (Chairman)	Date
//signed//	14 Mar 2014
Maj Kenneth Laviers, PhD (Member)	Date
//signed//	14 Mar 2014
Gilbert L. Peterson, PhD (Member)	Date

AFIT-ENG-14-M-13

Abstract

This investigation introduces a novel approach for online build-order optimization in real-time strategy (RTS) games. The goal of our research is to develop an artificial intelligence (AI) RTS planning agent for military critical decision-making education with the ability to perform at an expert human level, as well as to assess a players critical decision-making ability or skill-level. Build-order optimization is modeled as a multi-objective problem (MOP), and solutions are generated utilizing a multi-objective evolutionary algorithm (MOEA) that provides a set of good build-orders to a RTS planning agent. We define three research objectives: (1) Design, implement and validate a capability to determine the skill-level of a RTS player. (2) Design, implement and validate a strategic planning tool that produces near expert level build-orders which are an ordered sequence of actions a player can issue to achieve a goal, and (3) Integrate the strategic planning tool into our existing RTS agent framework and an RTS game engine. The skill-level metric we selected provides an original and needed method of evaluating a RTS players skill-level during game play. This metric is a high-level description of how quickly a player executes a strategy versus known players executing the same strategy. Our strategic planning tool combines a game simulator and an MOEA to produce a set of diverse and good build-orders for an RTS agent. Through the integration of case-base reasoning (CBR), planning goals are derived and expert build-orders are injected into a MOEA population. The MOEA then produces a diverse and approximate Pareto front that is integrated into our AI RTS agent framework. Thus, the planning tool provides an innovative online approach for strategic planning in RTS games. Experimentation via the Spring Engine Balanced Annihilation game reveals that the strategic planner is able to discover build-orders that are better than an expert scripted agent and thus achieve faster strategy execution times.

For my lovely wife who has taught me that you can have it all, but not all at once. And to my three sons who have filled my life with purpose.

Acknowledgments

This work is a culmination of the efforts of Dr. Gary Lamont to inspire and challenge me to well define the RTS problem domain - thank you for the guidance, pushing me to dig deeper, and our frequent chats.

Jason M. Blackford

Table of Contents

Page

Page

List of Figures

List of Tables

Table Page

List of Abbreviations

ONLINE BUILD-ORDER OPTIMIZATION FOR REAL-TIME STRATEGY AGENTS USING MULTI-OBJECTIVE EVOLUTIONARY ALGORITHMS

I. Introduction

This research serves to enhance strategic decision-making education by advancing artificial intelligence (AI) in computer generated forces for real-time strategy (RTS) games. Real-time strategy games provide a problem domain that is characterized as uncertain, dynamic, partially observable, and stochastic [16]. These characteristics make RTS games well suited for real-time decision-making military education because these qualities also well characterize real-world military events. The following sections present our research goal and objectives.

1.1 Military Education

The military employs numerous methods to educate and evaluate personnel in areas such as job competency, planning, and decision-making. A large portion of military education encompasses leadership and critical decision-making. Common decision-making techniques military personnel are equipped with include the Observe Orient Decide Act (OODA) model [49] and the Military Decision-Making Process (MDMP) [25]. Using AI agents in an RTS game environment enables new capabilities for military education including real-time dynamic content changes, adjustable challenge levels, and tailored content to individuals based upon skill level [63].

1.2 Real-Time Strategy Games

The RTS genre dates back to 1982 with the Cytron Masters strategy game released for the Apple II. Cytron Masters places two opponents on a twelve by six grid-map. Players

engage one another as opposing commanders, gathering electricity from generators to build units to destroy the opposing player's army. By today's RTS standards, Cytron Masters is more like a chess match given the small grid-map and limited number of units to control. However, the idea of real-time resource gathering and unit creation managed by a player acting as a commander is a motif shared by most modern RTS games.

Today's RTS games play-out as large scale war campaigns (Figure 1.1). Players begin with limited supplies and no army. From the start of a match, players must quickly orient themselves to their surrounding environment, similar to what the OODA decision-making model teaches military personnel, locate resources and identify enemy positions. In general, players must manage the simultaneous collection of two to three resources throughout the game in order to build their armies and advance their army's wartime capability through technology advancements. In addition to combating opposing armies, the players must also manage their home base or township (depends on the RTS game). In general, this involves construction of houses/barracks to provide room and board for soldiers and workers, balancing the work efforts of worker units to satisfy resource needs, base defense, research, and construction of critical buildings. How the player chooses the actions or decisions to make as they race the opposing player to victory is determined by the player's strategy.

Figure 1.1: Balanced Annihilation: Yellow army versus Blue army.

Most RTS games have well developed strategies that the community of gamers agree upon. In general, these strategies take form because of the efforts of game designers to ensure balanced, challenging, and fun game matches. An example is the Starcraft community. Starcraft is an RTS series which currently possesses two installments: Starcraft: Broodwars and Starcraft II: Wings of Liberty. Broodwars is the first of the series and is greatly utilized by the AI RTS research community for designing and evaluating various AI algorithms for use in non-deterministic, partially observable environments [7] [16][65].

As mentioned previously, RTS players attempt to advance the skills and technology of their armies in order to over power their opponent. This technology advancement is defined in the technology tree of an RTS game. A technology tree is a large decision tree that determines the execution of a player's selected strategy. It establishes an ordering of

actions a player must take in order to build and advance their army. Therefore, once a player selects a strategy, they must proceed with execution of the strategy by taking actions in the ordering stipulated by the technology tree. This leads to the idea of build-orders.

In an RTS game, a build-order is an ordered sequence of actions a player takes to execute their chosen strategy. Strategies are not static, often times players will blend or modify their strategy as game-play advances; however, for illustrative purposes it is more concise to assume a player executes one specific strategy. A strategy is generally broken into a set of sub-goals. Between each sub-goal is a unique build-order to move a player from one goal to the next. A goal is related to training a large army, researching technology to improve army abilities or collecting resources required for advancing through the technology tree. By the end of a game session, a player has achieved a set of goals by executing a set of build-orders.

Strategy execution is a planning problem requiring two components: firstly, determining a good goal-ordering to ensure expert level execution of a strategy with respect to resource allocations and time to execute the strategy; and second, issuing and executing actions in a build-order to ensure expert or near-optimal player performance- again measured in resource allocations and time to reach a goal. With respect to providing an artificial agent with the ability to rationalize across this planning problem, the first component involving goal-ordering can be determined by deriving a goal-ordering from the replays of an expert RTS player [67] [45] and then provide this to the agent via case-based reasoning (CBR). Resolving the second component is known in the RTS research community as the build-order optimization [22] problem and is the focus of this work.

1.2.1 Near-Optimal Build-Orders in RTS Games.

This investigation includes the development of an agent that optimizes strategy execution in order to perform at an expert human level; therefore, we make the assumption that the build-orders generated by an expert player are near-optimal. The objective of our

planning tool is to produce build-orders that are as good as or better than expert build-orders or near-optimal. Rather than use the term ”optimal” to describe the build-orders produced by our planning tool, we instead evaluate our build-orders to how much better they perform versus a human expert level build-order. The comparison of our produced build-orders versus expert level build-orders is conducted relative to the objective functions of our build-order problem formulation presented in chapter 3.

1.3 Research Goal

Our goal is to develop an AI RTS planning agent for critical decision-making education. This goal is motivated by an overall vision to build an RTS agent and an RTS game framework to evaluate a player's leadership and decision-making abilities.

1.4 Research Objectives

To achieve our goal the following research objectives are defined. Italicized objectives are research objectives that have been completed by predecessors to this current effort. The bold objectives, including 1a,2a and 2g, are the areas of contribution of this work:

1. Construct a complete player model of an RTS player to be utilized by an RTS AI agent.
   (a) **Develop (design, implement and validate) an online capability to determine the skill level of an RTS player**
   (b) *Develop an online capability to classify the strategy of an RTS player[61].*
2. **Develop an AI RTS agent to perform interactively at an expert RTS game playing level using opponent's skill level**
   (a) **Develop an online RTS multi-objective AI planning algorithm for generating human expert level RTS build-orders [22]**
   (b) Determine interactively RTS player's game-playing strategy

(c) Generate human expert level RTS tactics

(d) Dynamic content injection into RTS game

(e) Adjust difficulty settings based upon player's dynamic skill level

(f) Tailor feedback to player in real-time to enhance critical decision-making education

(g) **Integrate on-line planning tool with an agent in the Spring RTS game engine Balanced Annihilation game [3] and validate via simulation.**

1.5 General Approach

The approach taken toward developing our planning agent is derived from current research efforts in the area of AI for RTS games like Starcraft, Wargus, and Balanced Annihilation. More specifically, AI techniques designed to produce agents that compete at an expert level in the RTS domain. Utilizing a multi-scale agent approach [27] our focus is on optimizing the performance of the strategic decision-making process of an RTS agent. The emphasis is on producing a planning tool that is on-line and provides an agent with build-orders that are equal to or better than expert level build-orders. This is achieved through a combination of simulation, a well defined approximation model of the build-order problem, an MOEA, and CBR.

With respect to the first bold objective, **1a**, a linear transformation function that measures the speed of an RTS player's decision-making process with respect to a known strategy is developed. This transformation function reduces the number of features required for a classifier to classify player skill level [9]. The approach provides a high level description of a players skill level relative to a set of known player skill levels. This enables the possibility for the use of a hierarchical skill level classification approach for RTS games, with this technique acting at the highest level and more granular approaches being utilized at lower levels. This work builds from the RTS strategy prediction work established in [67].

For objective **2a** the RTS strategic decision-making process is framed as a build-order optimization problem. The build-order problem is modeled as an MOP [24] with three objective functions and three constraints. The mathematical model of the strategic decision-making process is then integrated into an RTS game simulator capable of executing strategic decisions common to generic RTS games including Starcraft: Broodwars [32] and Balanced Annihilation [3]. The simulator scores build-orders based upon the three objective functions of the build-order MOP. To generate build-orders, an MOEA framework in conjunction with the RTS strategic decision simulator is utilized to score a population of randomly generated build-order plans. To enhance the discovery of near-optimal solutions, expert solutions are injected into the population of the MOEA framework utilizing CBR. This results in an RTS strategic planning capability that is online and provides an agent with near-optimal build-orders. The capabilities of the strategic planner are demonstrated in the Spring game engine with the Balanced Annihilation RTS game satisfying research objective **2g**.

1.6 Thesis Overview

The following chapters present the accomplishments of this effort. Chapter two provides a background on current AI techniques utilized for RTS agents and other computer generated agents. In addition it provides a look into current approaches for measuring the skill level of an RTS player as well as a discussion on MOEAs. Chapter three discusses our approach to developing a strategic planning tool and contrasts it with other build-order techniques. Chapter four describes our experimental approach and the results obtained. Finally, chapter five provides discussion on future work, and how our current approach can be enhanced. **Our work is the first online build-order optimization technique to model the build-order problem as an MOP, and to use an MOEA to produce near-optimal build-orders for an RTS agent.**

II. Background

2.1 Introduction

This chapter introduces key concepts for understanding the work presented in this investigation. It starts with the employment of RTS games by the military as training and education tools. Then various AI frameworks are also presented which provide descriptions of research that has greatly influenced our work. In addition, several player modeling techniques are discussed in order to establish how to build an RTS player model for analyzing player performance. Following player modeling is an overview of a multitude of RTS planning techniques that motivated our research and impacted our design decisions. Lastly, a discussion on MOEAs and multi-objective problems is presented in order to introduce key concepts for understanding the build-order MOP presented in chapter three.

2.2 RTS for Education

The United States Air Force has employed several types of RTS games for education and evaluation at Maxwell Air Force Base. These games supplement learning objectives for education programs like Air and Space Basic Course ASBC and Squadron Officer School (SOS) [4]. One wargame in particular is named Tactical Airpower Visualization (TAV) [5]. The game is utilized to reinforce application-level learning with a focus on the Air Force core competencies [2]. In addition, the game is utilized to evaluate a player/team's ability to perform crisis-action planning and decentralized execution [4]. For a presentation on the utility of games for education refer to [35] [56]. The question becomes how to use RTS games effectively for educating and evaluating the ability of military personnel to make decisions. Can such decision making models such as OODA and MDMP be integrated into an RTS to evaluate a player's decision making abilities? A challenge to integrating such evaluation into RTS games is the lack of subjectivity. In terms of leadership, it is

often said that a decision is better than no decision, so a player should be evaluated on not only the quality of their decisions, but the timing of a decision. Similar to evaluating the performance of artificial agents in a task environment, it is better to see a solution implemented, though it may not be the best, rather than no solution taken at all. This is often measured in terms of rationality. Therefore a mechanism for measuring a player's ability to behave rationally in an RTS game environment must be developed. At a high level we classify a player's rationality via skill level as expert, intermediate, or novice.

The first objective of our research requires measuring a players skill level or ability to rationalize in a simulated task environment with the intention of using this information for an AI agent. Before metrics are applied to a player, a model of the player must be developed. Player analysis enables the development of this player model [63]

2.3 RTS AI Frameworks

A goal of current RTS AI research is to develop an agent that performs at the level of an expert player. This section provides details on where current research techniques are in reaching this goal. First a definition or model of an expert player is introduced and then various techniques for approximating this model are discussed.

2.3.1 RTS Expert Player Competencies.

As discussed in [28] [27] [52] there are a set of competencies an expert player is expected to master. These competencies include strategy execution, tactical maneuvering of units in battle, resource collection, production of units, buildings and upgrades, and scouting enemy positions. Each of these are intertwined with one another in that they introduce competing goals and resource allocation needs. For example, production of units, buildings and upgrades requires effective resource gathering. Yet even with strong resource gathering and production, a player without an effective strategy or ability to tactically maneuver units in battle will lose. Therefore a player must balance these competencies as needed throughout the game. With all of these factors to consider, an expert player must

be decisive in their decision making and utilize units and resources effectively. Idle workers hurt the ability of the player to gather resources and poor planning severely hinder overall strategy execution.

In addition to the RTS player competencies, Weber [66] presents three capabilities of expert players in RTS games: estimation, adaptation, and anticipation. Estimation and anticipation relate to predicting an opponents next action based on an observed strategy the opponent is executing. Adaptation corresponds to an agent's planning mechanism in which an agent adapts its plans in real-time to cope with changes in an uncertain environment and opponent. These three capabilities enable an AI to learn from and reason about its world. Together the competencies and capabilities describe an AI framework that enables an agent to manage competing goals in an uncertain and dynamic environment.

The goal of AI RTS research is to develop a framework that enables an AI agent to execute the RTS player competencies and capabilities at an expert level. An RTS agent must satisfy two objectives:

(1) Manage competing goals/tasks across the various competencies simultaneously

(2) Cope with uncertainty

The RTS agent pyramid in Figure 2.1 presents a generic organization of a multi-scale agent framework. The foundation of the framework includes the RTS player competencies that are implemented with various AI algorithmic techniques. These techniques range from complex search algorithms [52] to scripted managers. Above the player competencies are the expert RTS player capabilities. The capabilities rely on the algorithmic techniques utilized to implement the competencies. The implementation of the competencies and a well structured framework to enable communication between managers entails the capabilities, which collectively achieve handling uncertainty and managing competing goals in an RTS environment.

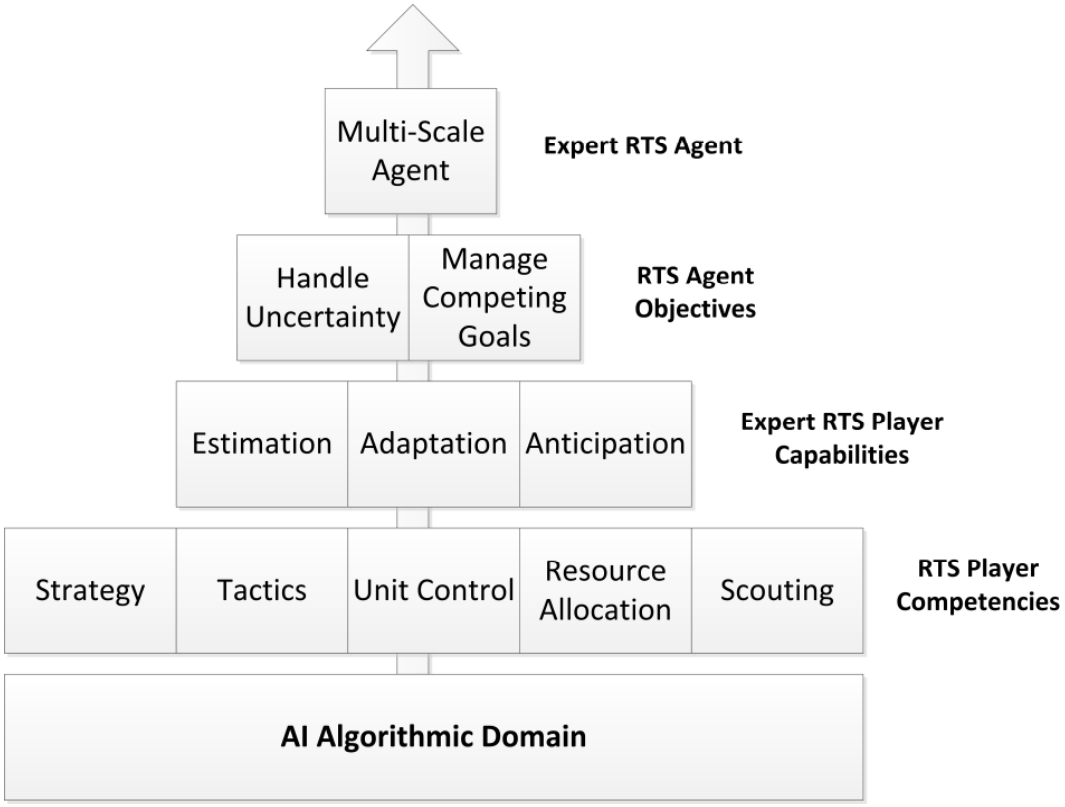

Figure 2.1: The expert RTS agent pyramid organizes from the bottom up what is necessary for constructing a multi-scale agent capable of playing an RTS game at an expert level.

2.3.2 Integrated Agents.

Various frameworks begin by abstracting the expert competencies into managers [27] [46] that aggregated together form an agent. The competencies may be divided into several managers including: strategy, income, production, tactics, and finally a reconnaissance manager. Each of these managers possess their own plans and goals that compete for resources and execution. One specific framework developed by Weber [66] utilizes reactive planning to orchestrate the competing needs of the various managers. The overall concept is that reactive planning enables an AI to operate across multiple levels of managers or competing goals simultaneously. From this a multi-scale AI is developed that reasons at different granularities by means of the various agent managers. The multi-scale agent

developed by Weber is implemented in A Behaviour Language (ABL) [69] which is designed specifically for reactive planning.

A final approach toward integrated agents are cognitive architectures. Cognitive architectures are modeled after the cognitive psychology model of human intelligence. In other words, the way humans process information through long term and short term memory. Two prominent architectures include ICARUS [38] and Soar [40]. The authors of [73] develop an RTS AI agent built around Soar and open source real-time strategy game engine (ORTS). These cognitive techniques are known as unified agent architectures. The subsections that follow present a multitude of algorithmic techniques for implementing the various competencies depicted in Figure 2.1. These techniques might be incorporated into a multi-scale agent as the core of various managers.

2.3.3 Reinforcement Learning.

The fundamental concept of reinforcement learning (RL) is exactly what the name stipulates, an agent learns over time the best actions to take in order to reach a goal state by repeated experience in the environment it operates in or task environment [51]. Every state an agent reaches includes a set of feasible actions available to the agent for execution. Each action is assigned an award based on how well it moves the agent toward a goal state. The objective of RL is to optimally reach a goal state by progressively discovering an optimal mapping of agent-environment states to agent actions. This mapping of states to actions is known as a policy. Given some reward function across all feasible actions, the optimal policy maximizes the reward from an initial state to a goal state. Unlike supervised learning [57], RL enables an agent to learn from its own experience rather than reference a set of rules on how to behave. Challenges with RL include tuning parameters to balance exploration versus exploitation and incorporating suitable domain knowledge for an effective reward function. RL has been utilized in RTS games to determine the best action an agent should take given information on an opponent [39]. It has also been

incorporated with CBR for transfer learning agent learns to perform unforeseen tasks based on knowledge from related previous tasks [54].

2.3.4 Scripting.

There are two variations of scripting: static and dynamic [72]. Static scripting is the simplest form of AI in that an expert encodes knowledge to an agent as a set of rules. Essentially every possible scenario an agent may encounter must be hard coded to the agent - as an event occurs the agent reacts. This takes the form of a sequence of conditional statement checks or a very simple finite state machine. The problem with this technique is that an agent does not reason or adapt to new information in the environment - ultimately the AI becomes predictable. Dynamic scripting overcomes the limitations of static scripting by incorporating a mechanism similar to RL. Dynamic scripting brings in a scoring system that enables an agent to score the effectiveness of its actions versus a player. This feature enables an agent to reason to a limited degree on which actions to take against a player based on historical data.

2.3.5 Case-Based Planning.

Case-based planning (CBP) is a planning technique commonly utilized in RTS games. It has been used for action prediction [39], determining build-orders [65] and plans [48], and extracting goals and sub-goals for agents [66]. This technique builds a library of cases derived from expert player game traces or player log data. These player traces, which are just a series of actions and time-stamps, are annotated utilizing an annotation tool to match action sequences with intended goals. These goals are defined by an expert in the RTS domain. Once a player trace has been annotated it is encoded into a set of cases that relate actions to goals. With a well developed case library an agent can determine the set of actions it should take to reach a goal state via a case selection function. Each case encodes a partial player plan utilized to achieve some goal state or execute a known strategy. This library of cases can be searched by an agent to determine the best actions to take to reach a

goal from the agents current state, or to predict the next move of an opponent player given a player model. As the agent searches the library, cases are scored heuristically on how well they lead the agent from its current state to the goal state. The case with the best heuristic value is selected.

Case-based planning provides an agent with a learning mechanism, enabling the agent to learn from experts. The authors of [45] present a case-based planning architecture designed to integrate real-time planning and execution for large decision spaces. Their architecture attempts to capture the expert capability adaptation by utilizing two modules in plan execution. One module referred to as the real-time plan expansion and execution (RTEE) module, keeps track of execution of current active goals in a tree structure. When RTEE comes across an open goal in the tree it queries the behavior generation module (BG) which returns a generated plan to reach the goal. This plan is derived from the current state of the agent and the expert case library. It is important to point out that the authors included in their case definitions the actual game state information for when the expert took the action. Game state information often relates to resource levels. Before RTEE executes the plan obtained from BG it sends the plan back to BG for adaptation. This last step is designed to take into account any new game state information that may have changed prior to RTEE beginning execution of a goal - keeps plans from becoming stale or obsolete. The authors implemented the system utilizing a reactive planning language similar to ABL.

Challenging aspects of CBP include generating game traces of expert players and then annotating these traces to formalize them as cases to be added to the expert case library. In general trace annotation requires defining a set of goals for the RTS domain the case library supports. Another challenge is determining the temporal relationship between goals in a single game trace. In [45] they contend that the RTS domain only requires a goal ordering based on sequential and parallel timing. Sequential meaning a goal 'A' must occur before

a goal 'B'. The author of [8] presents a temporal reasoning framework for establishing temporal ordering.

Lastly a case retrieval method or case selection function must be specified for the case base library. In [45] they use a nearest neighbor algorithm with a similarity metric that takes into account the similarity between the goal and game state. The case retrieval method greatly influences the quality of the case base reasoning mechanism as demonstrated in [65] and [70].

2.3.6 Monte Carlo Planning.

Monte Carlo planning takes as input an initial state and a goal state. It then iterates via simulation through a set of feasible actions defined for a state and scores state-action pairs with an evaluation function. Once the goal state is reached, the sequence of state-action pairs with the best score is returned as the plan of execution for an agent to reach the desired goal state. The quality of the solutions returned depends on an accurate state space abstraction and a proper evaluation function for action-state pairs. These items require expert knowledge of the domain being simulated in addition to an effective simulator; however, in the case of RTS games, the game itself may be utilized as the simulator. Monte Carlo techniques have been used for tactical assault planning [10] and strategy selection [21].

2.3.7 Air Force Institute of Technology AI Framework.

The framework currently being developed at the Air Force Institute of Technology (AFIT) attempts to capture expert competencies via several managers: strategy, tactics, build, economy, and defense. These managers are aggregated under one entity - the agent. The framework is implemented utilizing the programming language Python and interfaces with the Spring game engine. Spring is an open source 3D RTS game engine that provides several fully functioning RTS games. The game our agent is written for is Balanced Annihilation, which is a free modified implementation of the commercial Total

Annihilation game. Spring affords designers the ability to obtain all state information on players as well as the ability to fully design and implement custom RTS games. This is an important feature for experimentation and validation purposes.

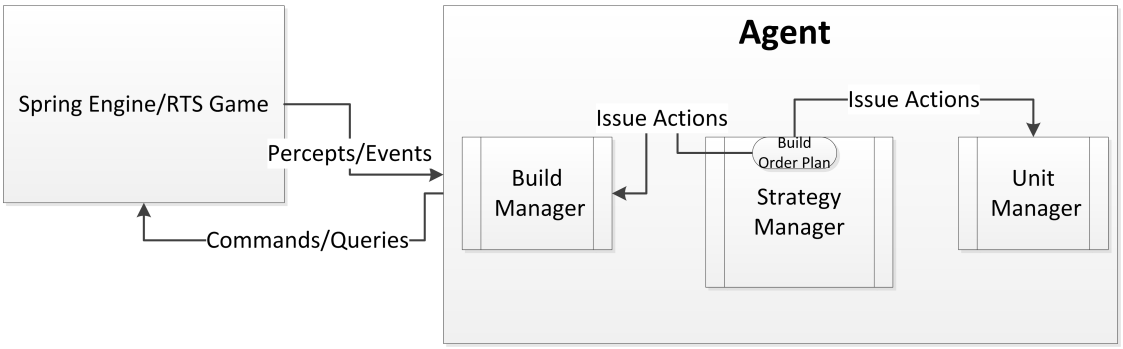

Figure 2.2: AFIT multi-scale agent architecture.

2.4 RTS Player Modeling Approaches

A player model is an abstraction of an actual player defined in terms of actions/events in a game. Once known, a player model allows for game designers to dynamically modify a game state according to what the player model infers such a player may enjoy. To achieve the research goal identified in chapter one, the RTS player model must define two components: a player strategy and player skill level. This section briefly discusses several approaches for player modeling through strategy prediction. These classification techniques define one-dimensional player models that relate to predefined game strategies for the specific games analyzed in the papers. The section ends with a discussion on determining player skill level.

Determining the strategy and tactics a player is executing versus an AI opponent is a great place to start for measuring the players understanding of simple strategic and tactical decisions. Such areas of examination include the players ability to engage an AI

opponent. For example, enemy tanks pitted against player infantry is a bad tactical decision. This poor decision could be the result of poor scouting, poor technology advancement, not understanding utility of units, poor strategic planning, or just poor maneuvering on the battlefield. Possible features for observing these types of player behaviors include the number of AI vs Player engagements, players use of scouts, battle wins and loses, player resources, player technology and upgrade advancements vs AI advancement, and the list goes on and on. The key point is that data must be collected on the player for evaluation. Collecting and analyzing such data is non-trivial. However, some insight into this is provided in the subsections that follow.

2.4.1 Hierarchical Player Modeling.

Hierarchical methods attempt to reduce the amount of computations for building a player model by separating classification into abstracted levels with the top tier classification layer being more general and the sub-layers more granular. The advantages of this approach are that different aspects of player data can be analyzed at the various levels allowing for unique classification techniques at each level. For example, in [53], the authors introduce a two tier hierarchical approach for classifying the player. The first tier utilizes a fuzzy classifier to distinguish a player from two play styles: aggressive or defensive. The classifier uses a single value to determine the player as either aggressive or defensive. The value quantitatively represents the amount of game-time the player spends attacking their opponent. The assumption here, is that the knowledge gained from this first tier, reduces the data to be examined at the subsequent classification tiers. Essentially, strategies available to an aggressive player will be different from those available to a defensive player. The final tier introduced uses the largest amount of information to determine the player strategy from the player style. Once a player is classified as aggressive, specific game events are listened for to aid in the final classification of the players strategy. The same goes for defensive play style. This final tier uses a technique that resembles reinforcement learning

in that a uniform reward is associated with each action taken by the player, probabilities are assigned to observations for likelihood of the observation relating to a predefined strategy, and a discount factor is included for emphasis on present actions in a finite horizon. The sum-product is calculated using these three parameters to obtain a confidence value for each strategy, essentially a maximum expected likelihood or confidence. The strategy with the highest confidence is selected as the most likely strategy being implemented by the player.

2.4.2 Machine Learning: A Supervised Approach.

The machine learning approach examined is the one introduced by Weber [67]. The premise of the paper is data mining or rather encoding player information from player logs into feature vectors, and then using supervised learning algorithms to classify these vectors with respect to predefined strategies or assigned labels. The importance of these vectors must be stated clearly. The vectors encode information on the players position in their technology tree or tech tree. In RTS games a tech tree is a decision tree that presents the player with technology advancements. These advancements include buildings that allow for production of units and weaponry, economic enhancements, advanced technology development or unit/building upgrades. How quickly a player progresses through this technology tree depends on their available resources that they have collected. Without these resources a player could be stuck at a low tech level while their opponent advances, putting the player in a vulnerable and weak position. Observing the decisions a player makes through this technology tree and the timing of these decisions provides information on the player's strategy. For example, it is reasonable to argue that a player who consistently chooses aircraft advancement over ground unit upgrades might be implementing a strategy that requires strong air assets. The authors collected several thousand player logs and built feature vectors from them. Each vector is then labeled as a predefined strategy using a set of rules defined by the authors. Any vectors that cannot be determined are labeled unknown.

Now knowing the strategies of the players, the authors utilized learning algorithms from the WEKA package [6] to attempt to classify the player vectors from various time intervals within the game. The learning algorithms attempt to classify the player from the feature vector as it is recreated over time simulating real-time, in-game classification of a player to build a player model.

2.4.3 Bayesian Model: An Unsupervised Approach.

The paper produced by Synneave and Bessi [58], takes an unsupervised approach to player modeling. Rather than attempting to classify a players strategy the algorithm defines a probability distribution across all possible tech tree decision paths a player could follow. These probabilities are calculated using observed events of build tree decisions, such as existence of a building and/or unit type, and timings of the appearances of these events. The possible build tree options were limited to observed replay logs from expert players, the same data set used in the supervised method previously discussed [67]. The advantage of this method is its ability to determine a player's strategy based on a confidence level. This means the system handles changes dynamically, being able to adjust to a player's strategy as it changes throughout the game. Whereas the supervised method determines strategies from a small pool of strategies defined by the authors, the Bayes approach assigns distributions across a large pool of possible technology tree decision paths. The authors propose that this method could be used to infer what decision the AI should make given the decisions of the opponent, but further research is required.

2.4.4 Measuring Skill Level.

Extracting player skill level in RTS games is non-trivial. Similar to the supervised approach approach presented in [67], features about the player's actions within the game must be obtained. The question then arises, what information might be useful to collect on the player to approximate their skill level. Avontuur [9] provides great insight into possible features for classifying a player's aptitude. One example provided by Avontuur

is actions per minute (APM), which relates a player's skill level to in-game mouse-clicks. Features like APM are classified by Avonturr as visuospatial attention and motor skills. The author further contends that experts perform better in this category of features; however, measuring better or rather the effectiveness of these actions is crucial. For example, a player issuing commands quickly, yet erroneously, will obtain a high APM. Determining the player's true APM requires a method for filtering irrelevant commands or mouse clicks.

The author uses an optimized support vector machine (SVM) known as sequential minimize optimization algorithm (SMO) to classify player skill level based on 44 features derived from player log data. A SVM is used for classifying nonlinear data sets. It uses a linear function (kernel) to transform the data into a linear feature space so that the data is linearly separable. Once a line is found that separates the data, support vectors for each class are used to maximize the separation between the line and the respective classes it separates. SMO is an optimized technique for using standard SVMs. It breaks the classification problem into a series of smaller sub-problems that are solved more quickly. For detailed information on SMO refer to [47]. The author groups the features utilized for classification into several categories including: visuospatial attention and motor skills, economy, technology, and strategy. The visuospatial attention and motor skills encapsulate the idea that an expert player has better motor skills and is able to observe more of what is occurring within the game, thereby making faster decisions. This idea is captured by filtering the APM. Filtering is used to ensure the effectiveness of the players actions to guarantee a high APM is a sign of skill and experience. The next category is economy which measures the effectiveness of the player to gain resources needed to advance in the tech tree as well as support defensive and offensive operations. The technology category is focused on capturing the total number of upgrades the player commits, research advancements the player engages, and their position within the tech tree. The final category is strategy which encompasses the types of units and buildings the player creates as well

as supplies used and gained. By performing some pre-processing on these features, the author trains their classification algorithm to determine the skill level of the player. The complexities of observing player skill level, as introduced by Avontuur [9], may be reduced by observing strategy timing. Essentially, a strategy dictates the actions a player will take. The speed of a strategies execution depends upon the ability of the player to perform resource gathering, defense, and base creation simultaneously. Therefore, a strategy schema as defined in [67] and its timing encapsulates the categories of features discussed by Avontuur. For RTS games, utilization of a strategy is necessary. A player cannot engage in the game without choosing a strategy of sorts. It is a shared idea in the RTS research community that the build tree of a player reveals their strategy [67]. A build tree is the parts of the technology tree the player has reached or executed. This is an assumption accepted in our work as well. Since a strategy captures the critical actions of a player, it is conjectured that the timing of the execution of a player's strategy is enough to rate the player's skill at a high level, meaning an expert, intermediate, or novice. Therefore, determining the second component of the player model, skill level, relies heavily on knowing the key features of the strategy the player is executing or a strategy schema. This eliminates the need for APM and filtering for skill level classification.

2.4.5 Summary.

The previous sections discussed building a player model around the strategy and skill level of a player. The benefit of such a model is that it enables an opponent AI to better rationalize within its task environment on how to engage the player. Knowing the player's strategy enables an AI to determine the best method of play against the player. However, in developing war game simulations or dynamic game play experiences, it is not enough to know the player's strategy. Knowing the players skill level reveals strengths and weaknesses in regards to the player's ability to think and operate both strategically and

tactically within a dynamically changing environment. For education this enables an AI to react appropriately with the player for the purposes of education and instruction.

2.5 Planning in RTS Games

There exist two areas of planning in the RTS domain: strategic planning and tactical planning [72]. Tactical planning aims to answer the question of how to engage the enemy. This requires an agent to have awareness of the map terrain (choke points), enemy controlled map locations, enemy unit strengths and weaknesses, and other pertinent tactical information. This area typically focuses on micromanagement of combat units to form large attack groups to engage enemy forces, and selecting the correct combat units to engage enemy forces based on the strengths and weaknesses of each unit [10].

Strategic planning encompasses a much larger area of operations than tactical planning. It is focused on implementing a strategy selected by a player. In general this requires macro-management or determining what and when to build, produce, and upgrade. However, there are others that present strategy execution with respect to battlefield management [52], we place this under tactical planning. In most RTS games a strategy directs the player through the RTS technology tree throughout the duration of a single game session. At a minimum, the operations falling under strategic planning include resource gathering, unit production, building construction, technology upgrades, and base expansion. Implementation of a strategy takes the form of a build-order. A build-order is an ordered set of actions or a plan to achieve a goal. For example, if a player seeks to obtain five combat units this involves a series of sequential and/or concurrent actions such as commanding workers to gather resources, constructing a barracks to build combat units, and then queuing the barracks to generate five units. Strategic planning attempts to generate build-orders to achieve goals. This thesis investigation attempts to address the issue of developing a strategic planner capable of producing build-orders that are as good as or better than expert build-orders. A more formal definition of the build-order optimization

problem is presented in the sections that follow. It is interesting to note that a strategic planning tool in an RTS game is essentially optimizing the size of the agents OODA loop or speed at which an agent executes decisions. In the military it is taught to minimize the OODA loop or the time required to observe, orient, decide and act. In an RTS game this is the build-order problem.

2.5.1 Behavioral versus Optimization Approaches.

It is important to address the two views of the build-order problem in strategic planning. The build-order problem is also the build-order generation problem. It is either represented as a behavioral problem or an optimization problem. We contend that the framing of the build-order problem greatly impacts the effectiveness of strategic planning tools. Ultimately the objective of any RTS agent planning approach must be to provide expert level build-orders online.

The behavioral approach tends to be more focused on producing build-orders that are directly derived or learned from expert RTS player build-orders. In general, this approach is implemented via case-base reasoning [65] [45]. The underlying assumption is that CBR will provide an agent with build-orders that resemble expert play. The advantages to this approach are that an agent is given the ability to learn good build-orders that will inevitably aid the agent in reaching a strategic goal. In addition the approach allows for online planning and execution. The disadvantage to a strategic planner limited to CBR is that it relies on cases derived from a finite set of expert data. This presents an implicit assumption that the experts are indeed generating near-optimal build-orders with respect to executing a strategy. However, there is no method or metric to validate this assumption.

The optimization approach looks at the build-order problem very differently. Build-order is a performance issue not a behavioral issue. The objective is to now equip an agent with a planning tool capable of performing at or better than an expert player - not mimicking behaviors. Objective functions are introduced to produce a planning technique

that searches for build-orders that optimize these functions. Via these objective functions a build-order produced by a planning tool is validated as equal to or better than an expert build-order. In addition, expert build-orders are utilized as a starting point for the search not an end point. Some concerns with this approach are related to computational time and its ability to produce online planning tools. We address build-order generation as an optimization problem and demonstrate an online capability.

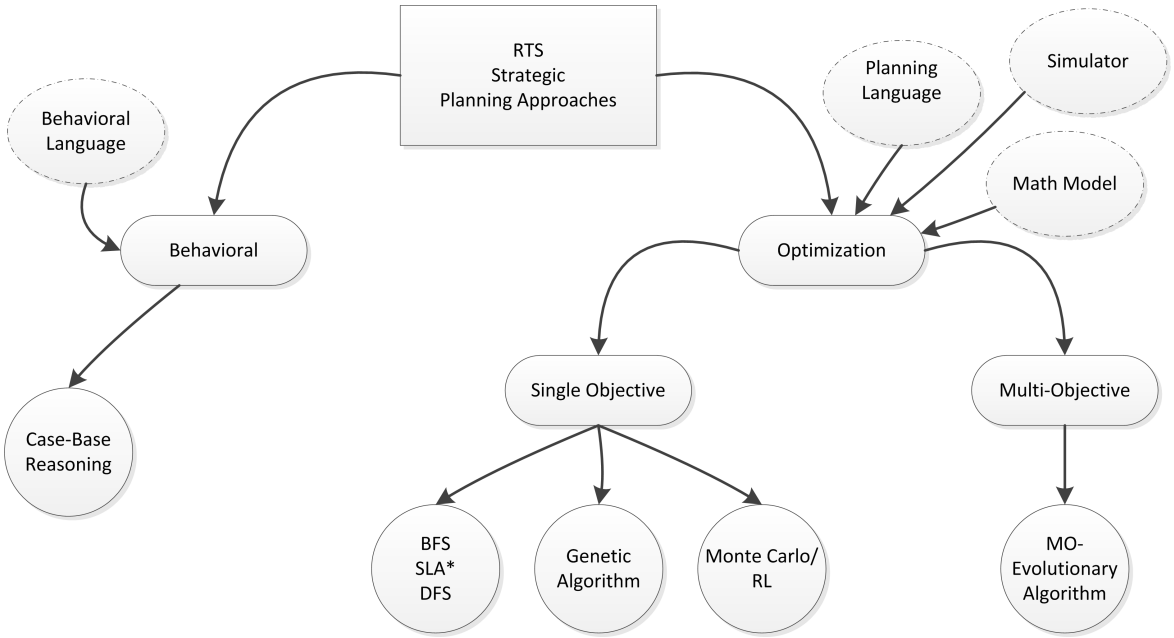

Figure 2.3: Build-order Approaches and associated algorithmic techniques. The dotted ovals represent additional components that are commonly associated with a particular approach, however, they are not inclusive or exhaustive.

2.5.2 Build-Order Optimization Techniques.

For this investigation we frame the RTS build-order optimization (BOO) problem as a planning and scheduling problem. Build-order optimization consists of determining the actions necessary to reach a desired goal state and then organizing the execution of these actions to minimize the makespan of plan execution. Build-order optimization seeks to find

an ordered set of actions to accomplish a given goal while minimizing the time to reach the goal. The build-order optimization problem takes as input a set of feasible or infeasible actions A, a goal state G, and an initial or current game state S. Each action a_i within A has resource preconditions the current game state S must satisfy before the state issues the action - which generates the child or next game state. Each action produces effects that are applied to the game state in which the action is completed or its duration expires. The current game state is defined by the values of current resource levels and the current game time in seconds. In terms of output, it is an ordered sequence of actions whose cumulative effects are captured in a final state S_f that is equivalent or satisfies the G requirements.

Several techniques have been developed to provide approximate solutions to build-order optimization. One particular formulation of the BOO problem is discussed in [20]. The authors of [20] present an online planning tool for optimizing resource production in RTS games. In particular they utilize the game Wargus [60]. Their planning tool attempts to solve both the planning and scheduling components of the BOO problem specifically aimed at resource production. The planner consists of two components: a sequential planner which uses means-ends analysis (MEA) to find a plan from an initial game state to a goal state with the minimum number of actions, and a heuristic scheduler which reorders actions in a sequential plan (output of the sequential planner) to allow concurrency and minimize the plan's makespan. MEA recursively solves sub-goals from an initial state until finally reaching the goal state. Sub-goals are derived from the goal state and sub-plans are formulated to solve each sub-goal, combining plans as they overlap.

The issue with the build-order problem model used in [20] is that it relies on the assumption that the best plan is the one that creates the minimum amount of actions and renewable resources in order to optimize makespan. However, focusing on a single objective does not capture a realistic model of an RTS game - RTS games are inherently multi-objective. With respect to generating the minimum amount of actions it is shown

by example that this is a poor assumption if concurrency of actions is not properly considered. For example, if a planner 'A' generates five actions that are not concurrent versus another planner 'B' that generates ten actions with pairs of actions being concurrent, then according to the underlying assumption of the model, 'A' is better than 'B'; however, with concurrency if the completion time of 'B's plan is the same as 'A's plan, then 'B' is superior with respect to effects or resource production. However, this is not explicitly formulated or stipulated as an objective of the model developed in [20]. The model has no means of determining whether 'A' or 'B' is optimal with respect to time and production of resources. As shown by example, the number of actions in a plan given duration is not an optimal feature of a build-order plan. Rather the timing and effects of the plan taken together determine optimality.

The second requirement for optimality presented in [20] is that the minimum amount of renewable resources is generated; however, in an RTS game this is not always desirable. The long term effect of producing the minimal amount of required resources to reach a goal will suffocate the player's economy in the long term. For example, the question arises of which is better, producing five workers to mine minerals to reach a goal or producing ten workers to mine minerals to reach a goal? With respect to action duration to minimize makespan producing five workers is optimal; however, this is a single optimal feature of an RTS plan, the effects of the actions must also be considered. Makespan is important but there is an underlying tradeoff surface between makespan and action effects. The authors of [20] adapted their approach in [19] which improves their heuristic scheduler by incorporating a bounded best-first search technique to better address the tradeoff surface of makespan and action effects.

Churchill and Buro in [22] utilize a depth-first search (DFS) branch and bound algorithm to perform build-order search. As input they take in a state S. DFS then performs a recursive search on descendants of S to find a state that satisfies a given goal G with the

edges in the graph representing actions. DFS uses a linear amount of memory dm where d is the depth the algorithm explored and m is the size of a node. For online learning, DFS affords the ability to end the search at any time and return the best found solution so far. The authors successfully integrate their online optimization planning technique into their Starcraft: Broodwars agent as presented in [23].

To reduce the complexity of the search space Chruchill and Buro [22] introduce several abstractions into their simulator. A few of these abstractions include:

1. To increase the speed of state transitions the authors assign a constant value for resource gathering rates derived empirically from professional game observations. It also eliminates gather resource actions which enlarge the complexity of the build-order search space for branching search algorithms like DFS.

2. To remove worker reassignments and further reduce the search space, the authors assign a preset number of workers to newly constructed resource collection sites like gas refineries in Starcraft.

3. The authors also inflate construction action durations in order to compenstate for travel times to build locations.

The following list outlines the manner in which the authors define and reduce the complexity of the search space examined by DFS:

1. First the authors determine the legality of actions for a given state. These rules are be found in [22].

2. To eliminate null states (states in which the player takes no action and is waiting for previous actions in other states to complete) from the search space they introduce the concept of fast-forwarding simulation. This concept is captured by A state transition function. Fast-forward simulation is important in order to reduce the computation time required to search the partial solution space for a good solution in real-time.

3. The authors introduce two heuristics for pruning nodes based on the heuristic evaluation of the path length left to the goal. They call these the "LandmarkLowerBound(S,G)" and "ResourceGoalBound(S,G)" heuristics where S and G are the current state and goal state respectively.

4. They also utilize breadth limiting to limit the branching factor of the search. This bounding issue is also addressed in [19].

5. The authors introduce the use of macroactions to further reduce the search space. When a macroaction is executed it is equivalent to executing a single action multiple times.

Another build-order technique is presented in [14] which uses MEAPOP to produce a partial-order plan (POP) of actions to achieve a desired resource goal. They then use SLA^* to schedule the actions. This algorithm is based on the A* search technique and learning for real-time purposes. They formulate the problem domain around resource production goals. MEAPOP generates a partial-order plan. POPs reduce the search space of traditional planners because they don't attempt to assign order to the actions taken in the plan. The plan produced is only required to satisfy constraints between the actions (dependencies) - order of execution is not considered. Therefore the search space contains plans that are not permutations of the same actions.

Any single objective metaheuristic (s-metaheuristic) [59], like DFS or BFS, requires action bounding to limit branching during search, especially global search techniques like breadth-first search and depth-first search. For example, in the game of Wargus a peasant requires an available town center to be constructed. To prevent the planner from constructing a town center for each new peasant the authors of [19] bound the number of command centers to build. This requires more expert knowledge to be inserted into the planning tool parameter tuning. This problem is also identified in [22]. In addition, single

objective functions fail to capture the tradeoffs made throughout play of an RTS game. An algorithm that capable of handling multiple objective functions must be utilized in order to capture the inherit tradeoff surface of RTS games. A better search technique is an MOEA that generates a surface that includes any number of objectives providing a decision maker or agent with options for deciding what is best for a given state of the game session. In addition, DFS and BFS do not keep track of good building blocks or action sequences. Both DFS and BFS generate partial solutions as they search for a goal state. This means each plan is uniquely constructed which is a waste of computational time. In MOEAs, the search space consist of complete solutions (feasible and/or infeasible) and the highest scoring plans are reused to build new plans which speeds convergence of the search.

2.5.3 Stochastic Plan Optimization with Case Based Reasoning.

The approach presented in [48] applies stochastic search to optimize the planning of CBR agents. The technique presented by the authors is divided into two stages both of which are similar to the selection and reproduction search components of an evolutionary algorithm. The first stage is plan generation which consists of two phases. In the first phase of plan generation, a population of base plans are derived from combinations of expert plan data and randomly generated plans. A trust value is assigned to generated plans which scales higher for plans with more expert derived components and lower for plans with more randomly generated plans. This trust value is utilized for exploration and exploitation when optimizing the generated base plans - diversity. The second phase attempts to iteratively optimize the population of base plans using various plan generation operators. This second phase is similar to evolutionary algorithms which use operators like crossover and mutation for reproduction. Similar to mutation and crossover, these operators are utilized to balance exploration and exploitation of the search space via probability parameters on the operators.

The second stage of the planner is an evaluation stage. It utilizes a heuristic to score plans similar to a fitness function as used in evolutionary algorithms. Plans with a high

score are selected and utilized for the next iteration of population generation. The heuristic function is an aggregated sum of multiple objectives. Aggregated into the heuristic are the following components: territory control; current supply of resources; military unit's resource costs are scaled by the current health of each respective unit (damage from enemy) and a constant scalar indicating worth of military units; the same is done for civilian units without the constant scaling factor; finally technology also without the constant scaling factor. The final score of a plan is the sum of the player's score minus an enemy's score. The heuristic is designed like a zero-sum or min-max heuristic in that the plan that ensures the maximum tradeoff of the players score at the expense of the enemy's score is favored.

Experimentation with this planning tool was mainly at the tactical level. However, it would be possible to utilize this technique for build-order optimization. Once again, this planner is a single objective optimizer; therefore, it does not capture the tradeoff surface present in RTS games. In addition, this technique is essentially a single objective evolutionary algorithm whose building blocks are plans, and whose initial population is constructed from experts.

2.5.4 Optimizing Opening Strategies.

Gmeiner, Donnert and Kostler the authors of [34] and [33] developed a multi-objective approach to compare already computed near-optimal build-orders discovered by a Starcraft 2 optimization tool known as EvoChamber developed by a third party. The authors of [34] provide a means for evaluating near-optimal build-orders in terms of economic and military power. However, the authors note the need for a pure multi-objective approach for determining near-optimal build-orders to obtain the approximated Pareto optimal or non-dominated solutions in a single run. This is interesting because our work provides a multi-objective approach that the authors of [34] can now use to determine the effectiveness of our planning tool.

2.5.5 Coevolution for Strategic Planning.

As presented in [11], Ballinger utilizes a coevolution system for discovering winning strategies against static opponents. Their current implementation utilizes a teachset defined by eight opponent strategies. These strategies are build-order plans that consist of a sequence of only feasible actions. Plans contain at most 13 actions and are encoded by three bits each. Their chromosome representation is a binary string of 13 actions or 39 bits. Plans consist of a limited number of build actions for the creation of units and buildings and a single attack action. They utilized a scripted tactical planner in which once an attack group is generated they are issued to attack the opponents home base and any enemy units along the way.

Ballinger attempts to optimize a single aggregated objective function provided below. The objective function attempts to optimize the ability of a strategy to achieve three goals. The first is to spend as many resources as possible under the assumption that the more resources being spent means the stronger the units and structures being produced (recall that this is very different from the assumption made by [20]). The second is to optimize the number of enemy units destroyed, and finally to optimize the number of enemy buildings destroyed.

$$F_{ij} = SR_i + 2 \sum_{k \epsilon UD_j} UC_k + 3 \sum_{k \epsilon BD_j} BC_k \tag{2.1}$$

The value of F_{ij} is the fitness of player i against opponent j. The first goal is observed by SR_i which is the total amount of resources spent by player i. The second goal is captured by summing the unit cost of destroyed opponent units represented by UD_j where UC_k is the cost to build unit k. The final goal is realized by summing the building cost of destroyed opponent buildings represented by BD_j where BC_k is the cost of building k.

The search components of the coevolution algorithm consist of a roulette wheel for selection, single point crossover, and bit-flip mutation. They also included a fitness sharing mechanism designed to ensure diversity in the population of solutions.

The limitations of this design is that to perform evaluation of solutions requires playing through an entire game session against all opponents defined in the teachset. In addition, the opponents are well defined - no uncertainty the coevolution system has complete information of the enemy strategy. This system does not satisfy the expert capability of being adaptive nor can it be used as a real-time planning tool. This makes their technique an offline learning tool.

It is interesting to note that Ballinger uses case-base injection for a establishing a teachset. The author takes a complete strategy or build-order plans of a human player and uses these to train the coevolution algorithm offline. The plans are not directly injected into the population of the coevolution algorithm. Another limitation to this technique is that it only looks at very small build-orders - currently limited to 13 actions. This is a very small search space for experimentation considering the actual size of an RTS search space.

2.5.6 Case-Injected Genetic Algorithms for RTS Strategies.

A case-injected genetic algorithm (GA) for optimizing attack strategies is presented in [42]. The author developed a custom RTS game called Strike Ops. The paper presents a genetic algorithm player (GAP) designed to optimize a plan of action or strategy and execute it in real-time. GAP is presented as an adaptive and online planning tool.

There is a subtle difference in the meaning of strategy for Strike Ops compared to the various RTS games presented discussed so far. Essentially Strike Ops consists of two teams Red and Blue. Blue plays the game by sending aircraft to attack Red's ground assets which are buildings, and static defensive and offensive weapon systems. Both Red and Blue seek to do the most damage to the opposing side while minimizing damage to themselves. GAP plays as the Blue attack force and therefore models the problem of determining an attack strategy as an optimization resource-allocation problem. The model consists of two parts: first, which assets to use on which targets and second, how to route platforms to carry out the allocation determined in the first component. The author presents this as a trade-off

surface that describes a good allocation versus a good route where a good allocation might optimize the damage to the enemy but require a more risky route and therefore also increase damage to Blue's own units. The optimized objective function is:

$$fit(plan) = Damage(Red) - Damage(Blue) - d * c \tag{2.2}$$

where d is the total distance traveled by Blue's aircraft and c is some constant to scale the penalty as a result of d. The fitness function favors shorter routes, more damage to Red and less damage to Blue. This function is very similar to a zero-sum game function [50] with an added penalty.

The author goes on to demonstrate that by injecting expert human strategies into GAPs population, the algorithm finds better strategies. These expert strategies are injected via case-injection which resembles case-base reasoning. Expert actions are recorded and then transformed into chromosomes for injection into GAPs population. This added mechanism speeds up computational time to adapt plans to the enemy opponent in real-time and also increases the ability of GAP to play like a human. It is interesting to note that GAP has a tendency to over optimize solutions with respect to the fitness function leading it to lose valuable human strategy information. The over optimized strategies tend to perform worse than the human strategy. To resolve this issue the author utilized fitness inflation to increase the fitness of solutions that contain human strategies. This bias' the search so that GAP keeps knowledge gained from human players.

2.6 Multi-objective Optimization Problems

Popular pedagogical optimization problems include the Knapsack problem, graph coloring, and maximum clique. In general, these problems have a single objective function to maximize/minimize with some constraints. In the case of the Knapsack problem the objective is to maximize the total value of items contained in a knapsack with the constraint that the weight of the objects contained in the knapsack do not exceed the weight capacity

of the knapsack. This problem is easily modeled utilizing linear programming techniques as presented below [59]:

Given a knapsack of capacity B and a set of items I with weights and values of w_i and v_i,

$$max \sum_{i \in S}^{|S|} v_i \tag{2.3}$$

with constraint,

$$\sum_{i \in S}^{|S|} w_i^t \leq B \tag{2.4}$$

where $S \subseteq I$ or the items contained in the knapsack.

The knapsack problem becomes a multi-objective optimization problem by introducing additional objective functions. A popular multi-objective version of the knapsack problem is introducing a second knapsack with a unique capacity. Unlike single objective problems, MOPs no longer have a single solution but multiple solutions that describe a tradeoff surface defined by the objective functions. These solutions are defined as Pareto optimal solutions. A solution is Pareto optimal if no improvements can be made on a single objective of the solution without degrading the other objectives. A formal mathematical definition of MOP is presented below [59]:

$$max/min\ F(x) = (f_1(x), f_2(x), ..., f_n(x)) \tag{2.5}$$

given any number of equality or inequality constraints, and where $n \geq 2$ and x represents a vector of decision variables. A popular algorithmic domain for solving MOPs is MOEA.

2.6.1 Multi-objective Evolutionary Algorithms.

Evolutionary algorithms (EA) are a population-based metaheuristic (P-metaheuristic) [59], which unlike local search algorithms such as depth-first search, start with a population of solutions to search. The search components of an EA include selection, reproduction and replacement. The selection component is used to determine which individuals of a population to select for reproduction and sets the selection pressure of an EA. Reproduction

is used to generate children. Common operators for reproduction include mutation and crossover. The final component of EA's is replacement. This component determines which members of the original population (parents) and newly generated children will be kept for the next iteration of selection and reproduction.

MOEAs utilize the same search techniques as single objective EAs. The major difference is that MOEAs return multiple solutions across the objective functions being optimized. The advantages of MOEAs are their ability to discover more than one member of the Pareto optimal set (P^*) per iteration. In addition, depending upon the diversity of an MOEA's population, an MOEA is less susceptible to the continuity of the Pareto Front. Finally, MOEAs are capable of addressing both search and multi-objective decision making.

Some disadvantages to MOEAs is that ensuring the effectiveness of the solver requires effective modeling of the MOP to be solved. Representing individuals or solutions of the MOP improperly will greatly deteriorate the ability of the algorithm to converge. In addition, much thought must be given to each search component to ensure diversity and the right amount of selection pressure. Poor diversity is often times a result of high selection pressure - elitism often results in premature convergence to a local optimum rather than a global optimum.

In [29] the authors present that the size and shape of the Pareto optimal front depends on the number and interactions of the objective functions. Deb introduces the terms 'conflicting' and 'cooperating' when describing objective function interactions. Conflicting functions have extreme differences in optimum solutions and function values; whereas, cooperating functions have similar optimum solutions and function values. Often times conflicting functions contribute to the discovery of large Pareto optimal fronts. With respect to MO-BOO the objective functions are cooperating.

With respect to the number of decision variables present in an MOP, it is reasonable to assert that the difficulty of the MOP will increase. Generally speaking, constraint-satisfaction problems become more cumbersome and expensive to solve (in terms of time) as the number of decision variables increases. This is due to constraint handling Increasing the number of decision variables, introduces new constraints, both between decision variables and on the decision variables themselves. As more constraints are introduced, the solution space becomes smaller and possibly discontinuous. As identified by Deb [29] constraints may hinder an MOEAs ability to converge and maintain diversity.

As presented in [24], there are three tasks that MOEAs must do well: move toward the true Pareto-front, maintain diversity on the Pareto-front, and preserve good solutions. The characteristics of objective functions strongly affects how well an MOEA performs these three tasks. Ultimately the difficulty of an MOP depends upon the characteristics of the objective functions and the interactions between those functions. Selecting the correct MOEA to solve an MOP is non-trivial. The No Free Lunch theorem (NFL) [74] reminds designers that there is no one algorithm to solve all problems. The problem domain must be carefully evaluated.

2.7 Summary

This chapter has introduced the five competencies of an RTS player and the three capabilities of an expert player. It has also provided discussion on a multitude of algorithmic techniques, developed by the AI research community, for encompassing the competencies and capabilities in the managers of a multi-scale agent architecture.

Furthermore the chapter establishes the significance of player analysis applied to RTS games in order to generate player models providing agents with information on their opponent. The player models consist of two pieces: player strategy and player skill level. This model can then be utilized by an agent to enhance its goal formulations and planning processes.

Planning in the RTS domain consist of strategic planning and tactical planning. The focus of this work is on strategic planning. Chapter two points out the importance of properly framing the build-order problem and presents two perspectives on how to frame the build-order problem. The first perspective, behavioral, frames the build-order problem as a behavioral issue, whereas, the optimization perspective views build-orders as a performance issue.

III. Methodology

3.1 Introduction

The goal of this thesis investigation is to develop an AI agent with the ability to perform at an expert level for critical decision-making education. This goal is apart of a vision encompassing dynamic content injection, adjustable difficulty settings, and tailored feedback in real-time to enhance decision-making education. The development of this capability relies on the three objectives outlined in chapter one - **Objective 1a, 2a and 2g**. This chapter presents how those objectives are achieved. The first section discusses our methodology for building an RTS player model - encompassing skill level. Section two presents our approach to strategic planning for RTS game agents, and the final section covers the integration of our agent into the AFIT framework presented in chapter two. The agent framework with the Spring engine, enables the possibility of an AI entity to take the information provided by the player model and utilize it with our strategic planning tool to dynamically interact with the player.

3.2 Building the Player Model

The RTS player model is an important abstraction or approximation of how the player is perceived in the game. The first component of the RTS player model is strategy. The second component of the player model is skill level. The focus of this effort is to determine skill level from a known player strategy.

3.2.1 RTS Player Skill Level.

The first research objective involves acquiring a large dataset of skilled and unskilled player data from a known RTS game domain. Due to the great availability of player data for the Starcraft: Broodwars game, we selected Starcraft: Broodwars for developing our skill level technique. In particular, Weber provides expert player log data via [64]. The data

captures game logs for the nine permutations of race matches in the Starcraft: Broodwars game. The data consists of player vector that capture a player's strategy for a single game session. Each vector is labeled with a corresponding strategy number that is determined by a rule set defined by Weber. Weber identifies six strategies for each race. For more details on the data set reference [67].

In Avontuur's investigation on RTS player skill level [9] the author presents a design and results for a player skill level classification technique utilized for the Starcraft RTS game. The disadvantage to this method is that it requires large amounts of player data that are not necessarily available during a single game session. In addition, the technique is very specific to the Starcraft games. Due to the limitations of this classifier, and the limited amount of research in the community on skill level classification in RTS games, our work presents a new perspective on addressing this problem.

There is one critical assumption for our technique to work. First off, the initial expert data must capture a wide spectrum of expert strategies, but not necessarily all. As presented by [9] experts exemplify motor and visuospatial skills that exceed those of other skill levels. As such, the decision process of experts is faster. A conjecture can be formulated that states by knowing a player's strategy and the timing of the actions to execute this strategy, a player's skill level with respect to strategy execution can be determined relative to experts executing the same strategy. Fortunately, Weber's data captures this exact information. Now having the strategies of hundreds of experts and their timings for executing these strategies, it becomes possible to define an expert space that captures a large variety of expert strategies. Unlike [9] we contend that there is no need to incorporate all the feature categories the author identifies because a player's strategy and timing of strategy encapsulates all of these categories. For example, knowing how much a player has collected in resources is unnecessary for classifying a player at a high level since the strategy and timing of actions of a player depends upon resources. This assumption simplifies the pre-

processing and analysis. It is also important to state that there are several factors that influence an experts decision to execute a strategy. These factors cannot be ignored when comparing the strategy of an unknown player to a known expert player. The factors are: opponent's selected avatar race (which is either Protoss, Terran or Zerg for the Starcraft series), map level terrain, and the player or agent's own selected avatar race. These factors are aggregated together to describe the game scenario. When evaluating an unknown player to an expert player the game scenarios should be the same. However, if provided a range of experts on various maps with various strategies then it is expected that the unknown player can be evaluated against the distribution or space of experts.

Our skill level evaluation design is based upon the assumption that experts, with respect to beginner players, are able to plan and execute strategies quickly. We attempt to capture the speed of execution of a player strategy by calculating the average distance between decisions (DBD). Therefore, our conjecture is that an expert will have a low DBD average compared to that of a beginner whose average should be high relative to a given expert DBD for the same game scenario. Figures 3.1 and 3.2 are histograms that depict the distribution of expert players with respect to DBD.

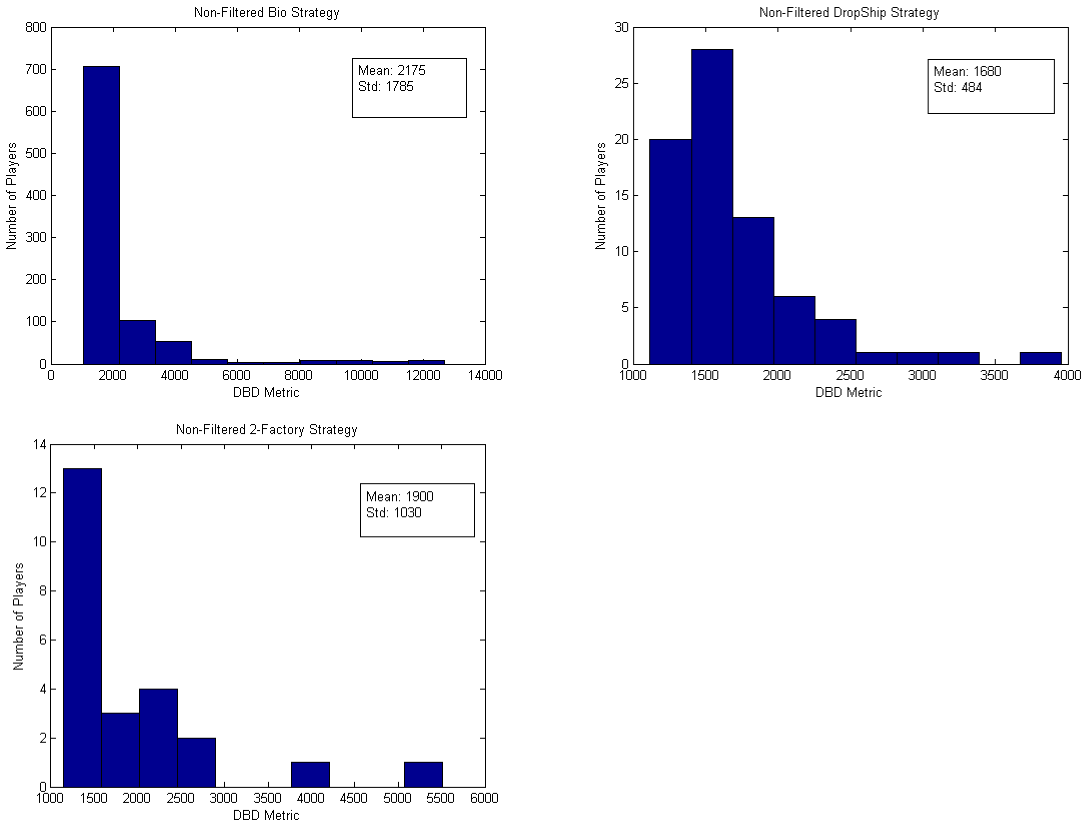

Figure 3.1: Distribution of DBD metric for expert players of Starcraft: Broodwars. This data is prior to filtering for outliers.

These histograms capture three strategies out of the six strategies experts execute when playing as Terran vs Zerg for the Starcraft game. The figures clearly capture a range for DBD that experts predominately operate in across strategies. The DBD metric not only allows for determining a players skill level, but also provides a metric for measuring the performance of the strategic planning tool versus known expert players.

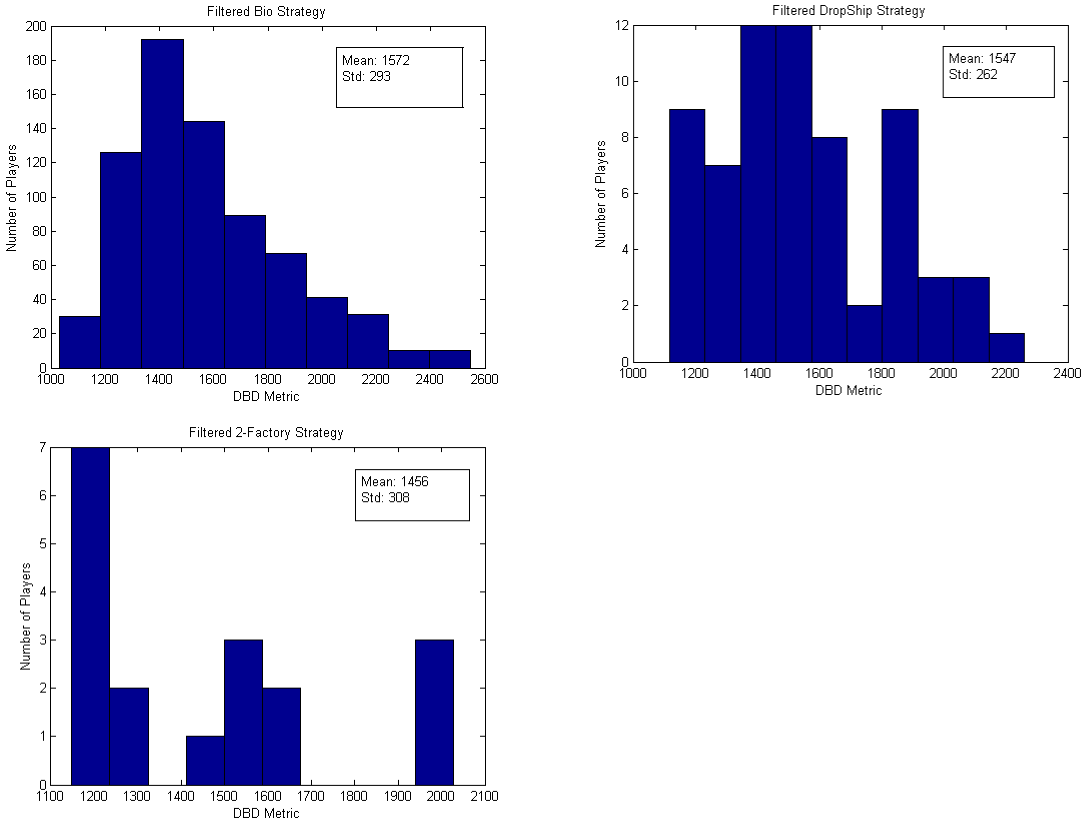

Figure 3.2: Filtered for outliers, these histograms depict the distribution of the DBD metric for expert players of Starcraft: Broodwars.

The following strategies Bio, Dropship and 2-Factory depicted in figure 3.1 were selected because they had the most player data of the six strategies for Terran vs Zerg. These histograms were calculated by applying the DBD linear transformation to the original raw dataset of expert players provided by Weber. From these histograms it became apparent that there were some significant outliers in the dataset. To remove these outliers we filtered the dataset based on a minimum number of decisions that a player must have made while executing a strategy during a single game session. The filter was applied in order to remove players from the data that appeared to correspond to non-expert players. The filter removed players who made less than twenty decisions out of the fifty

possible decisions with respect to the strategy decision schema utilized by [67] for strategy prediction and classification. The histograms for the filtered data are shown in figure 3.2.

3.2.2 Measuring Skill Level: Distance Between Decisions.

The linear transformation applied to the player vectors is below where E represents the length of the original player vector or the number of features in the player vector, and N is the number of features the player did not make. The features capture player decisions during game play.

$$\frac{\sum_{i=1}^{E}(D_{i+1} - D_i) + 1000 * N}{(E - N)} \tag{3.1}$$

The problem with averaging is features with a value of zero will drive the average down, thereby making a unskilled player appear highly skilled for being indecisive. To combat this issue a penalty of 1000 game cycles for each zero, or non-decision, present in a player's vector is added. Then only divide by the total number of decisions the player actually made. The reason for utilizing 1000 game cycles is that it empirically appears to be slightly higher than the average window between decisions. In addition, it provides the best spread of separating expert from non-expert data as depicted in the histograms in Figure 3.3. In Figure 3.3 the histograms depict a set of known experts in blue with a small set of known non-experts in red. By adding the penalty of 1000 cycles, the separation of the data is very clear. This small example only serves to validate the utility of a penalty. By increasing the penalty the distance between the two sets greatly increases. The penalty value is dependent upon the RTS game being analyzed. The value utilized for Starcraft will most likely not be acceptable for a different RTS game.

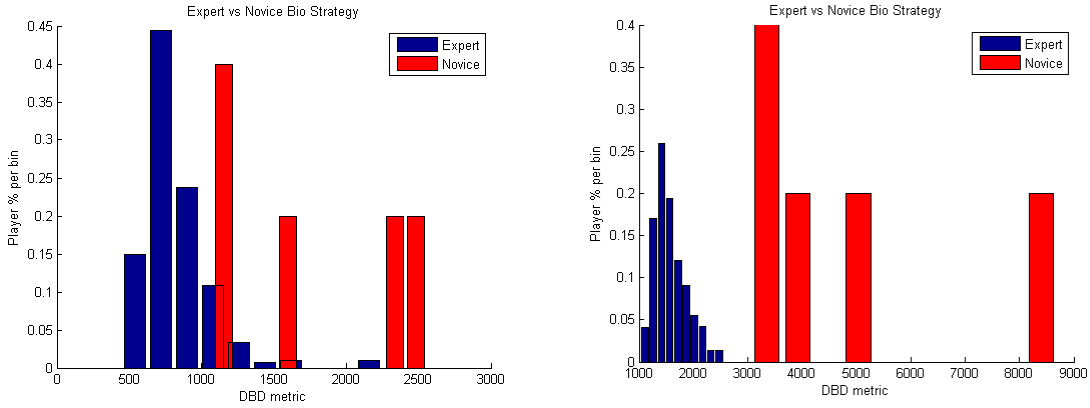

Figure 3.3: The histogram on the left demonstrates the separation without a penalty. The histogram on the right applies the penalty.

3.2.3 Utility of Skill Level.

In the Starcraft game, players can compete against each other over Starcraft game servers. Skill level is utilized to ensure players combat other players that can play at their skill level. This is to ensure a fun and challenging game experience. It should be obvious that to engage players and keep them playing, a game must be challenging and fun, however, more on key characteristics of player engagement can be found in [35]. These same two characteristics are also important for engaging students in education [35][56]. With the vision of our work encompassing RTS games for decision making education, skill level provides insight into the players experience [63].

A player model provides information to game designers and to artificial agents on how the human player is experiencing the game. The RTS player model consists of two components: strategy and skill level. It has been demonstrated in various works the utility of developing expert AI that can identify and incorporate information on a human opponent's strategy [67]. However, with respect to education, skill level is an important piece. Knowing a player's skill level an agent can adapt the difficulty level of the game experience to ensure the player is challenged but not overwhelmed. Challenge is important

for keeping students engaged [35]. In addition, our skill level metric enables online classification of a human player which can be used to provide feedback to the player on how well they are making decisions toward executing their strategy. Feedback is a crucial component to education and any game developed for education must provide students with feedback [35]. Our skill level metric measures how well a player handles the workload of an RTS game. Recall, that the skill level metric DBD is measuring the time between key milestones of a known strategy schema. A strategy schema clearly defines player tasks and goals. These actions and goals are the workload of an RTS player. Over time, the DBD metric provides insight into how well, relative to prior knowledge of an expert DBD distribution, a player is managing tasks within the game.

Dynamic content injection is the act of changing the game play experience as a player is experiencing the game. Knowing the skill level or workload of a player enables the possibility of injecting additional in-game scenarios to challenge a player's decision making process under stress. For example, in the Age of Empires RTS game, if a player engages in the single-player mode of the game, they play through numerous RTS game scenarios. In one case, the player is tasked at the beginning of the game to construct a home base and a small unit of troops for defense. However, unexpectedly the player is then informed of a rescue mission or resource sites that they must beat the opposing AI agent too. The player must now switch from developing their home base and defenses to initiating a rescue mission or reconnaissance team to the target site.

3.3 Strategic Planning Tool

The second objective of our research is to develop, implement and validate a strategic planning tool that produces build-orders for an RTS agent that are as good as or better than a human expert. The planning tool must produce near-optimal build-orders in real-time. Many planning problems are of complexity P-Space complete [17], but the complexity of the build-order problem is yet to be determined. Given that it is a planning problem and a

scheduling problem it is at least NP-complete [18] [**?**]. Therefore it is necessary to use an approximation technique to search through the decision space of the build-order problem. For this reason we selected an evolutionary algorithm. To achieve a real-time planner our design utilizes an optimization approach that includes a mathematical representation of the build-order problem with constraints, a simulator to evaluate action plans along side the RTS game, and an MOEA able to search the solution space for good build-orders. To provide the agent with expert goals and goal-ordering CBR is incorporated. Our approach is similar to the CBR techniques discussed in [70] [45]. Case-based reasoning provides an agent with goals to plan toward and expert goal-ordering relative to the agent's current game state.

3.3.1 Analyzing Optimization Approaches.

As presented in chapter two, there exists at least two types of optimization approaches: single objective and multi-objective. To model the build-order problem a multi-objective approach is taken in order to build a high fidelity model of the RTS problem domain. It is easier to justify this selection by demonstrating the disadvantages of the alternative approach.

The decision to approximate a problem as a single objective or multi-objective problem depends on the complexity of the problem to be solved - generally determined by the number of competing objectives and constraints [29]. In general, most real-world problems can be viewed as attempting to optimize competing objectives with multiple constraints [59]. With respect to the RTS problem domain, we contend that it is a problem domain with multiple competing objectives and constraints that can be described by a trade-off surface. The RTS build-order trade-off surface is defined at by time and resources. The build-order problem pertains to strategy execution. We conjecture that the faster a player is able to implement an appropriate strategy against an opponent, the better the player's performance against their opponent. In general this is the underlying RTS expert

assumption that experts are capable of implementing strategies faster than intermediate and beginner players [9].

Implementing a strategy is by definition reaching a list of goal states. An effective plan will empower the player to reach goal states in the minimal amount of time required. In RTS games these plans are build-orders. By organizing the actions within a build-order a player can increase or decrease the time to reach a goal. Immediately it becomes apparent that time is an objective. However, not as obvious are the requirements necessary for reaching a goal state in a timely manner. These requirements are resources which we describe as unary and volumetric resources. This manner of characterizing RTS game resources as unary and volumetric resources with producer/consumer constraints is derived from the work of [71]. Therefore, new objectives are derived in terms of resource amounts and allocations. For an experienced human player these objectives take the form of questions which are what, how and when to utilize a resource. For a computer agent, these questions must be defined mathematically. It is our assumption that a single-objective function cannot adequately approximate the decision making process of an expert RTS player.

3.3.2 A Nonconstraining Decision Space.

To ensure diverse and near-optimal build-orders are found, steps must be taken to allow for a large decision space. The assumption is that a larger decision space better reflects the RTS planning domain that expert players operate within. To allow for this requires the design of a robust RTS build-order simulator architecture with the ability to simulate all possible strategic decisions a player can make in a RTS game. Our simulator supports this design requirement. Essentially an agent utilizing our tool is not limited to a small subset of actions, they have the option to execute any action defined in the RTS game with respect to strategic decision making. This is unlike other evolutionary and genetic algorithim techniques discussed in [11] [42] which limit decision making. In fact, the action strings or accepted build-orders of our planning tool can range from very small

(one action) to very large (200+ actions). Often times constraining the search space is an attempt to ensure only feasible action strings are considered by the optimization algorithm [34]. However, this as a great limitation to empowering an AI with the ability to execute strategies at an expert level because it abstracts away too much of the decision space of RTS games. This abstraction limits the size of the decision space and diversity of solutions a search algorithm can discover. Therefore, by allowing our planning tool to consider infeasible solutions this removes constraints on the decision space and improves diversity of solutions. By incorporating a repair mechanism our planning tool is able to consider infeasible solutions, correct them and output feasible build-orders with associated fitness values.

3.3.3 Modeling Build-Order Optimization as Producer/Consumer Problem.

In formulating the mathematical model to approximate the build-order problem, a goal programming approach was adopted [24]. Goal programming is defining objective functions that minimize a distance to a goal. Distance is defined loosely as any quantitative metric that can be utilized to measure equivalence of a feasible solution from a multi-objective problem population to a target goal.

This paper adopts the nomenclature introduced in [20] which describes two categories of resources: renewable and consumable. The authors of [20] further characterize these two categories of resources in the RTS domain as one of the following four types: Require, Borrow, Consume, or Produce.

(1) **Require**: a resource that is persistent throughout the execution of the action but can be shared by other actions at the same time. For example, multiple units collecting wood and returning it to a lumber mill as in Age of Empires. The lumber mill is a resource that can be used by multiple units executing the collect wood action simultaneously.

(2) **Borrow**: A resource that is persistent throughout the execution of an action but cannot be shared. Is limited to one action at a time. For example, a barracks can only produce one

military unit at a time. Other requests can be queued, but it will only perform one action at a time.

(3) **Consume**: A resource that is used or decremented at the start of the execution of an action for a given game state. To build a home requires 100 wood. This amount is deducted from the total amount of wood present in the game state at the time of the request.

(4) **Produce**: A resource that is produced at the end of the execution of an action or the effect of an action. This amount is added to the game state.

Renewable resources include production buildings and units (workers/military) and are either required or borrowed by actions. Consumable resources are items like gold, wood, and food and are consumed by actions. Resources cannot be both renewable and consumable. In the operational research (OR) community renewable resources are known as unary resources and are formally defined for RTS games as a resource of capacity one as presented in [71]. This implies that any two activities requiring the same unary resource cannot be scheduled concurrently or overlap in execution. In OR consumable resources are labeled volumetric and represent a collection of resources [71]. Therefore, concurrency of actions, depending on a volumetric resource's availability, is possible. With these definitions of resource, it is possible now to identify the constraints that apply to scheduling activities requiring unary and/or volumetric resources in RTS games.

A solution to a scheduling problem attempts to minimize the makespan of a set of jobs - in the case of build-order optimization, actions. In combination with planning, a new problem forumlates which results in determining a plan to reach a desired goal state and then minimizing the makespan of the execution of this plan. The conjecture that the BOO problem is a planning and scheduling problem with producer/consumer constraints is first addressed in [71]. In RTS games, this scheduling problem consists of two parts: Scheduling highly cumulative activities and scheduling highly disjunctive activities. Both pieces are required to minimize the makespan for executing and completing actions to

reach a desired goal state from an initial state. Cumulative actions are actions that can be executed or issued concurrently (overlap) on the same resource. This applies to actions requiring volumetric resources. Disjunctive scheduling consists of pairs of actions that cannot be executed concurrently on the same resource. This applies to actions requiring unary resources. It is possible for actions to require both unary and volumetric resources; however, in the RTS domain an action will most likely only require one unary resource, but may require multiple volumetric resources. For example, in the popular AOE games a soldier requires a barracks (unary) and food or wood and gold which the latter are all volumetric resources.

3.3.3.1 Cumulative Scheduling.

In cumulative scheduling, activities are constrained by volumetric resources. The constraint is the cumulative producer/consumer constraint [71]. The cumulative constraint requires that the sum of the amounts of consumable resources required by a set of actions scheduled at a time t does not exceed the amount of the resource available at time t. For example, if it costs 100 food and 15 gold to produce a foot solider in an RTS game, and it costs 200 food and 100 gold to research advanced armor for the foot soldier, then to execute these actions concurrently, there must be a total of 300 food and 115 gold at the time of execution of these activities. Notice that these activities are not constrained by unary resources between eachother. To produce a soldier requires the unary resource of a barracks and to research armor requires the unary resource of a blacksmith (both are buildings). With respect to RTS games and volumetric resources in RTS games a slightly modified representation of the cumulative constraint [26] can be expressed as:

$$\sum_{a_j \in A}^{|A|} r_j^t \leq L_t \tag{3.2}$$

Where t is the time of request of the set of actions contained in the action set A, and a is a member of the set of actions being executed at time t. The amount of a single resource (gold, food, or gas) required by an activity a_j is expressed as r_j. The total amount or

volumetric quantity of the resource available at time t is represented as L. Keep in mind that more likely than not an activity will require several resources, but this expression only captures the constraint between actions on a single resource. This constraint must be satisfied for all volumetric resources shared between actions executed at a time instance t.

3.3.3.2 Disjunctive Scheduling.

It is important to clairfy what a unary resource is in an RTS game. This discussion is derived from the definition of a unary resource in RTS games from [71]. At first it would seem that the number of workers can be considered a consumable resource. For each action that requires a worker, simply assign a worker to the action. However, individual workers themselves are a resource, so it is better to divide the non-unary resource (total number of workers) into unary resources (individual workers)[71]. In the RTS domain each non-unary resource of capacity C is divided into C subsets (one capacity per subset), and each activity requiring the non-unary resource is divided into C sets and assigned one of the unary resource's subsets. For example, if $C = 15$ workers then there exists 15 subsets of the unary resource worker. Possible activities requiring a worker include: mining 100 gold, cutting 100 wood, or building a structure which has a constant duration. Each one of these actions requires holding onto the unary resource for some constant duration that is generally fixed and known (derived empirically via a simulation or available data). Therefore with respect to unary resources in RTS games, each action consists of a duration of execution and a domain of possible start times. Note that this assumption of having known action durations does not hold in all RTS games including games like Total Annihilation and Balanced Annihilation. How this is handled is discussed in later sections.

A popular technique for performing disjunctive scheduling is constraint satisfaction programming (CSP) [13]. The objective in using CSP to schedule activities under a unary resource constraint is to reduce the set of possible values for the start and end times of pairs of activities sharing the unary resource. A precedence ordering is establish for actions as

follows. As presented in [12] given two actions A and B that both require the same unary resource, schedule them according to the expression below:

$$end(A) \leq start(B) \vee end(B) \leq start(A) \tag{3.3}$$

This expression shows that A precedes B or B precedes A. A solution is found when the assignment of domain start times satisfies the Boolean expression above. However, satisfying the above expression generally requires the introduction of additional constraints such as priorities, release dates and due dates.

In RTS games it is uncommon for a single action to require more than one unary resource. Therefore it is assumed in this work that an action requires at most one unary resource. It is possible for actions to require multiple volumetric resources and a single unary resource - it is assumed that an action requires at most two volumetric resources. In addition, actions possess known durations, but do not have explicitly required start or end times (again this applies to most mainstream RTS games). A domain of start times can be formulated to minimize makespan once a build-order plan is generated. In general, actions requiring unary resources must first satisfy the cumulative constraint before being considered for disjunctive scheduling. Whereas actions requiring only volumetric resources are not subject to disjunctive constraints. Cumulative scheduling influences the manner in which actions can be considered for disjunctive scheduling. The cumulative constraint states clearly that an action cannot be executed if the required volumetric resources are unavailable. Therefore the start time of an action is a function of the time required to gather the necessary volumetric resources for action execution. The disjunctive constraint becomes an issue when two actions satisfying cumulative constraints seek concurrent execution, but require the same unary resource subset. Therefore cumulative scheduling can be viewed as a release time of an action - in otherwords an additional constraint to disjunctive scheduling. The disjunctive constraint stipulates an ordering in time between actions, but does not consider minimizing makespan. The disjunctive constraint is satisfied

once the boolean relationship between the start and end times of a pair of actions competing for concurrent execution is satisfied.

A final constraint that is present in RTS games is the exist constraint. This constraint is implicit to all RTS games and is derived from the technology tree of an RTS game. Essentially it stipulates that an action A can only be executed if a unary resource R exists. This is different from the disjunctive constraint, in that the action A does not require use of the resource R, only that the resource exists. For example, in Starcraft before a barracks can be constructed a command center must exist. This is not the same constraint imposed by a barracks being required to build a marine. A way to distinguish the disjunctive and exist constraint is that a barracks is used or borrowed by the action to build a marine. Essentially a unary resource cannot be both an exist and disjunctive constraint for a single action to be executed. When we mention that a unary resource is required by an action we mean with respect to the disjunctive constraint not the exist constraint. The exist constraint can be formulated logically as the implication statement:

$$B \implies A \tag{3.4}$$

Where B and A are actions. This expression draws a logical implication between actions B and A - B implies A. For example, in Starcraft the unary resource *CommandCenter* must exist before a unary resource *Barracks* can be produced. Therefore, the action *BuildBarracks* is B and the action *BuildCommandCenter* is A. An alternative view is to bind the variables B and A to the unary resources produced from the actions *BuildBarracks* and *BuildCommandCenter*. From this alternative binding, the unary resource *Barracks* cannot exist without the unary resource *CommandCenter*. This implies that the action *BuildBarracks* cannot be taken without action *BuildCommandCenter* having already completed.

In considering the scheduling and execution of an action it must pass the three constraints presented above in the following order: exist, cumulative, and finally

disjunctive. For example, in the Starcraft game, to build a barracks the following ordering of constraints, phrased as questions, are: Does command center exist? Are there enough volumetric resources? Are there any workers (unary resource) available to build the barracks? The BOO problem is summarized by the three decision questions below.
(1) Given a set of actions as defined by the technology tree of an RTS game(ignoring feasibility), what various plans can be formulated to satisfy a goal state?
(2) Given a set of volumetric and unary resources, how can the exist, cumulative and disjunctive constraints be satisfied so as to minimize the makespan of a selected plan?
(3) What is the best tradeoff amongst a set of plans that minimize distance to the goal state and minimize makespan?

The statements above describe a tradeoff surface between distance to a goal and makespan. This tradeoff is due to feasibility of perfectly satisfying a goal state. Given infinite time it may be possible to reach a goal state, but the tradeoff is time. Due to the nature of RTS games being finite and dynamic, goals will change quickly and the best plan may be the one that gets the closest with the smallest amount of time. A tradeoff surface empowers a decision maker to decide what is more important. This work presents BOO as a three dimensional MOP with three constraints suited for MOEAs.

3.3.4 MO-BOO.

The first decision question on the multi-objective build-order optimization problem (MO-BOO) flags the issue of how to formulate a plan consisting of a set of actions to reach a goal state. The best way to formulate this problem is to understand the requirements of the goal and develop a metric to measure the quality of the effects of the actions taken. To drive the search toward the best action decisions requires incorporating domain knowledge with respect to the desired goal and action options. Two heuristic functions have been defined in [22] for this purpose. The authors of [22] utilize these functions as heuristics to guide a DFS-BT search through a constrained search space. They select the heuristic

with the worst value (highest value) to reflect a state in the search tree in order to ensure an admissible lower bound. The authors present these functions in loose terms, but a formal mathematical model is presented below.

$$\sum_{i \epsilon A_G}^{|A_G|} D_i * (\{G_i\} - \{S_i\}) \tag{3.5}$$

The actions of the goal state are contained in the set A_G with members i. The sets G_i and S_i represent the goal state and current state of the game respectively. The objective is to identify the actions in G_i that are not contained in S_i and determine the cumulative duration of these actions in G_i not in S_i. This heuristic provides a lower bound on the quality of the current state in terms of its distance from the goal with respect to the duration of actions not yet taken by S_i. With respect to the actions mentioned, i, what is really being examined are the effects of these actions. For example, if the goal state, G_i, contains a barracks and the current state of the game S_i does not, then this translates into an action build_barracks that must be taken to reach the goal state. All actions have a duration D_i.

The second heuristic captures the consumable resource requirement that the goal state has achieved compared to the current game state.

$$\frac{1}{C_r}[\sum_{r \epsilon A_G}(A_{G_r}) - R_{S_r}] \tag{3.6}$$

This expression translates as follows: given a set of actions in a goal state, A_G, each action has multiple associated consumable or volumetric resource costs represented by r. The current state has a total number of resources available whose amounts are represented by R_{S_r}. By summing the resource costs of actions in the goal state and subtracting the available resources in the current state, a duration can be calculated for how long it will take the current state to achieve the required consumable resources in the goal state. Each resource will have a rate of collection that is represented by C_r.

However, with respect to an MOEA these heuristics can become objective functions that describe a tradeoff surface which is how they are employed in this work.

$$min(\sum_{i \epsilon A_G}^{|A_G|} D_i * (\{G_i\} - \{S_i\})) \tag{3.7}$$

$$min(\frac{1}{C_r}[\sum_{r \epsilon A_G}(A_{G_r}) - R_{S_r}]) \tag{3.8}$$

For the first function the objective is to minimize the duration of actions not taken by state S_i. For the second function the objective is to minimize the difference in required and available resources in the current state for all volumetric resources required by the actions, or said in another way to minimize the time required to collect the volumetric resources needed to execute the set of actions in the goal state or A_G.

Ultimately the best values of these functions is zero. However, there is a competing need between the two objectives. One attempts to minimize the duration of actions not taken while the other attempts to minimize the difference in required and available resources. These objectives are commensurable in that they both measure time [24].

Finally a third objective function must be introduced in order to achieve makespan minimization. This objective function is formulated from literature as a modified scheduling problem [55].

$$min(\sum_{i \epsilon A}^{N} F_i) \tag{3.9}$$

Where A is the set of actions selected and F_i is the finish time of each action. This expression closely relates to the scheduling problem of using M machines to schedule N jobs, $M << N$. The difference is that not all actions in A compete for the same resources. Some actions in A can be scheduled concurrently due to different cummulative and disjunctive constraints, while others must wait for resources to become available due to disjunctive constraints. In scheduling terms this expression is minimizing the makespan or duration of the time to complete all actions in A. The only constraints that exist for

minimizing the makespan are those presented by the actions - exist, cumulative, disjunctive. Each action has several preconditions before it can be issued. In the case of Starcraft most actions will have no more than three resource constraints. Generally two of the resources are volumetric resources and the other is unary (i.e minerals or gas and building or worker). Once again, this objective function is commensurable with the preceding objective functions. In summary, the MO-BOO problem is formalized with three objective functions under cumulative and disjunctive scheduling constraints as well as an implicit exist constraint.

3.3.5 Representing MO-BOO in MOEA Domain.

The advantage to utilizing an MOEA to solve MO-BOO is an MOEAs ability to operate along side a simulator. Simulation is a more powerful problem solving technique than utilizing planning languages because planning languages are often times limited by scalability and ability to represent complex relationships in some problem domains; such as action concurrency or ”no moving targets” rule in RTS games [37]. A simulator overcomes these challenges by the very fact that an expert of a problem domain can generate data that can utilized by an MOEA to solve the problem. In addition an expert can ensure a simulator is able to overcome challenges suchs as ”no moving targets” by programming the simulator to handle these situations appropriately.

For our design and experimentation we utilized the Jmetal MOEA framework. This framework is freely available online and well documented [44]. The Jmetal framework provides the capability to generate new problems to be solved by the MOEAs included in its framework. Simply wrapping the simulator as a problem in Jmetal allowed for fast and easy integration and access to a multitude of MOEAs.

3.3.5.1 Search and Objective Space of MO-BOO.

The search space of MO-BOO consists of all possible strategic decisions a player can make in a specified RTS game. There are countably infinite action strings of finite

length that can be generated by a player in a single game session. Searching through this space to optimize the three objective functions is at least a combinatorial NP-complete problem The search space is not constrained to feasible solutions, infeasible solutions are allowed. A repairing constraint handling technique is utilized to make infeasible solutions feasible. Allowing infeasible solutions enables the potential for better solutions to be found by empowering the algorithm to explore more action plans. This constraint handeling technique is discussed shortly. Solutions or action strings are represented with two variable types. Both variable types include a string of actions and have the same cardinality. The first variable type is an array of integers, with a domain of values ranging from one to the total number of available strategic actions. A finite number of decision variables contained in the integer array are initialized to a random ordering of actions. The second variable type is a binary array. This array is the same size as the integer array and each decision variable corresponds to whether or not the action defined in the integer array will be taken or not. A one is used to enforce execution of an action and a zero is to signify not to take an action. Jmetal provides the ability to define unique solution types. This solution type was built from preexisting solution types in Jmetal and has been named the $ArrayIntAndBinarySolutionType$. Representing solutions in this manner enables the possibility for the MOEA to explore more action orderings by not constraining the solutions to be feasible when they are generated. As solutions are simulated in the simulator a repairing function flips the decision bits of individual actions depending on whether or an action was feasible (passed constraints). Therefore, an initially infeasible solution becomes a feasible solutions that now maps to objective space. This encourages exploration, diveristy and preserves good building blocks.

Objective space is three dimensional and defined by the three objective functions outlined in previous sections. The phenotype of an action string from solution space is a single point in the three dimensional objective space. The Pareto front is a tradeoff

surface between the three objectives. The domain of the objective space is positive real numbers which means the objective space is uncountably infinite. A large objective space allows for more diversity amongst solutions. The objective functions for each solution are calculated from a inputted goal state provided by the user. The goal state lists the desired volumetric and unary resources the player would like to reach as well as the combat units. The solutions attempt to reach this goal state and are not limited in time, but by their string length. An example of a solution mapping to objective space is provided below for clarification. This is not an optimal build-order just a simple illustration of the genotype and phenotype mapping. The action string is defined as the following:
Solution Representation: 7 1 5 7 0 1 5 7 3 6 1111101111

As derived from table 3.1 the solution representation string translates into: Build Barracks, Collect Mineral, Build Refinery, Build Barracks, Build Worker, Collect Mineral, Build Refinery, Build Barracks, Make Marine, Build Command Center. Based on the bit string only 9 of the 10 actions are taken. The one not taken is collect mineral at position 6 in the integer action string starting the index count at one.

Table 3.1: Simulator Available Actions: Minerals (min), Supply (supp), duration (secs)

Action	Duration	Vol Resource	Unary Resource	Action Num
Make Marine	20	Min,Supp	Barracks	3
Make Worker	20	Min,Supp	Command Center	0
Build Supply	40	Min	Worker	4
Build Refinery	40	Min	Worker	5
Build Cmd Ctr	120	Min	Worker	6
Build Barracks	40	Min	Worker	7
Collect Gas	n/a	n/a	Worker	2
Collect Min	n/a	n/a	Worker	1

The objective scores in Table 3.2 reveal the following about the solution selected: The first objective measured in at 120 seconds. This means that the solution had a lower bound of 120 seconds, based on the actions remaining to be taken, to reach the goal state. This is the best possible time remaining if the resource constraints are satisfied. The quality of this measure or how close to this lower bound the solution is relies on the second objective which measures the required resources the plan needed in order to reach the goal. A value of 102.22 reflects that by the time the plan ended it still required 102.22 seconds to meet the volumetric requirements necessary to satisfy the goal state. Therefore, the first objective is no longer a lower bound. The third objective measures the total time or makespan of the execution of the plan. In the end, this plan would have reached the goal state if more decision variables or the length of the action plan was increased. The goal state is listed in Table 3.3. Below the goal state table are also included the initial state Table 3.4 from which the solution starts and the final state reached by execution of the solution in Table 3.5.

Table 3.2: List of Objective Measures

Objective #	**Fitness Score**
1	120.0
2	102.22
3	786.67

Final Solution String:

Solution Representation: 7 1 5 7 0 1 5 7 3 6 0111101111

Notice that the initial action string was an infeasible solution because it required the first action to be taken as building a barracks, however, this action did not satisfy the minerals requirement of the initial state of the game, so the planner flipped the decision bit to zero to make the solution feasible.

The above objective measures were achieved utilizing a modified version of NSGAII from the Jmetal framework. More on this is presented in the next section.

Table 3.3: Goal State

Resource	Amount
Mineral	n/a
Gas	n/a
Command Center	2
Barracks	3
Supply	0
Marine	5
Worker	8
Refinery	2

Table 3.4: Initial State

Resource	Amount
Mineral	100
Gas	0
Command Center	1
Barracks	0
Supply	5
Marine	0
Worker	5
Refinery	0

Table 3.5: Solution State

Resource	Amount
Mineral	162.0
Gas	0
Command Center	2
Barracks	2
Supply	3
Marine	1
Worker	6
Refinery	2

3.3.6 Selecting the MOEA.

MO-BOO consists of a large number of decision variables with multiple constraints. Its objective functions are cooperating and commensurable. The domain of the objective space is very large. An interesting characteristic of MO-BOO is the possible relationships between the actions or decision variables. It is clear that any plan generated or feasible solution must obey the technology tree of the player's selected race - Terran in this thesis. Therefore, an MOEA that can identify these relationships would be advantageous. At the very least, any selected MOEA to solve MO-BOO must be able to effectively explore a large search space, be scalable with the number of decision variables and enforce diversity of solutions. Provided below are some of the MOEAs suitable for MO-BOO.

An algorithm capable of modeling the relationships between decision variables is worth while for MO-BOO due to the ordering of actions stipulated by the technology tree. This tree dictates which actions must occur first before other actions are possible. Therefore, there exists a natural ordering of actions present in RTS games that can be

exploited. An evolutionary distribution algorithm (EDA) such as the multi-objective Bayesian optimization algorithm (mBOA) [36] with a proper modeling technique (capable of modeling the complex relationship of actions within an RTS game) may be able to determine these relationships and provide insight into the ordering of RTS game actions.

Evolutionary distribution algorithms rely on the initial population to construct an accurate model of good solutions. The more accurate the model the faster an EDA algorithm will converge to a Pareto front. The goal of an EDA is to obtain high probabilities for the best solutions through an iterative process. The various implementations of EDAs can be identified by the specific probabilistic model utilized for search [31]. These models range from simple to complex (i.e. Population Based Incremental Learning (PBIL) to mBOA). It can be argued that the more complex the model the better the search [31]. The authors of [31] contend that the limitations of EDAs depends largely on the probabilistic model being utilized. This is because more complex models, such as Bayesian nets, are able to better determine or model the interactions amongst variables in a problem. This is not to say that Bayesian nets are the best model, utlimately the modeling technique utilized is dependent on the complexity of the problem domain - NFL. In the case of MO-BOO, it may be helpful to utilize known expert strategy data as an initial population to start the search. Scalability becomes an issue for EDAs in terms of the complexity of modeling relationships between the decision variables and the computational time required to determine them [31].

An MOEA attempting to solve MO-BOO will utilize a simulator for exploring the best action plans. Immediately this resembles learning a policy from utility theory [50]. Some possible MOEAs that cooperate well with simulators and utility theory are multi-objective Ant-Q (MOAQ) where the Q represents $Q-learning$ [41] and Multi-Objective Montecarlo Tree Search (MOMCTS) [62] and [15].

NSGAII was selected for its simplicity and scalability with respect to computational time. The advantage to utilizing NSGAII compared to the other approaches is that NSGAII

scales better in terms of computation time with respect to population size and decision variables. This scalability quality is a desirable feature for an online RTS planning algorithm to possess. In addition, NSGAII is well documented and utilized throughout the research community for a wide range of MOPs.

3.3.7 NSGAII in Jmetal.

The Nondominated Sorting Genetic Algorithm II (NSGAII) was developed by Deb, et al [30]. NSGAII is a non-explicit building block (BB) MOEA - it does not directly operate on or examine the BBs of a genotype [24]. Instead it implicity operates on BBs utilizing standard genetic operators.

The algorithm starts by instantiating a population of random individuals. Solutions are evaluated and then ranked and sorted based on nondomination. Essentially the rank of an individual relates to how many individuals the solution is dominated by with respect to the known Pareto front. The nondominated solutions in the population are assigned rank 1. Rank depths are established for each individual member of the population by removing nondominated solutions and ranking the remaining members. This continues until all individuals have been ranked. The algorithm then applies a selection routine to obtain parents and utilizes crossover and mutation to generate offspring. The union of the parents and offspring is evaluated, ranked and sorted with respect to nondomination. Elitism is applied to take the best members based on rank and add them to a new population. A crowding distance is also utilized to ensure diversity of the new population along the Pareto front.

The Jmetal framework provides an implementation of NSGAII based on the work presented in [43]. Their version utilizes a quality indicator, specifically hypervolume [24], to determine the convergence speed of the algorithm. The algorithm takes as parameters a population size and maximum number of evaluations. In addition Jmetal provides the ability to substitute various genetic operators into the algorithm for crossover and mutation.

The use of these operators depends on the solution type utilized by the problem specified (i.e. binary, real, etc). A distance object is also specified and used to calculate the crowding distance for diversity.

3.3.8 NSGAII for MO-BOO.

The following provides a description of the search components implemented with the NSGAII algorithm provided by Jmetal. In addition constraint handeling is discussed. The section concludes with a discussion on the crowding measure utilized.

3.3.8.1 Selection Method.

The selection method controls what is known as the selection pressure of an MOEA. Selection pressure falls into a continuum of high to low. A high selection pressure encourages more elitism or selecting the best solutions from a population for reproduction. This can commonly have the affect of driving the search to converge too quickly which violates diversity and preservation of good solutions - two of the three objectives of any MOEA [24]. A low selection pressure therefore does the opposite. It will typically preserve diversity due to a slow convergence rate. However, it may not preserve good solutions or ever converge if there is no pressure to select good solutions. Ultimately selection pressure relates to exploration of a search space.

To ensure a large exploration of the MO-BOO selection space, binary tournament with $k = 2$ was utilized with NSGAII. Binary tournament selects k members from a population and selects the best solution from the k members. The advantage of binary tournament is that it is simple and has low selection pressure compared to alternative approaches like roulette wheel selection, stochastic universal sampling and rank-based selection [59].

3.3.8.2 Reproduction Operators.

Immediately after the selection phase the reproduction phase begins. In NSGAII reproduction consists of the application of a crossover operation followed by a mutation operation. A discussion on the operators selected for MO-BOO is presented.

Crossover is utilized to ensure a child of two selected parents inherits the good genes of the parents. There are a multitude of variations of crossover, however, for this design single point crossover is utilized. Single point crossover takes two parents and produces two children by selecting a crossover point in the genotypes of the parents and swapping the genes of the parents to generate two new individuals or offspring. Associated with crossover is a crossover rate. This rate relates to how often the operator is performed on pairs of parents. Crossover exploits good solutions. In the case of MO-BOO crossover acts on the integer portion of the solution type or the actions.

Mutation is the final operator and is applied to the child generated by crossover. It takes a probability parameter that corresponds to the probability of modifying a single gene in the genotype of an offspring. The probability is generally set to $1/k$ where k represents the number of genes or actions in an individual. This generally results in at least one gene of an offspring being modified. The mutation operator utilized is bit-flip mutation. It acts on the binary string of the solution type which determines whether or not an action should be taken.

3.3.8.3 Constraint Handling.

The decision space of MO-BOO consists of feasible and infeasible solutions. To ensure convergence toward the Pareto front a mechansim inside the evaluation routine of the simulator was developed to correct infeasible solutions so that they become feasible. The mechanism simply switches the bit of an action that is designated by the simulator as not being a feasible action to take. The benefit of this mechanism is that the action is not lost. Depending on the ordering of actions or the availability of resources, an action may be infeasible for one solution, but possibly after a crossover operation and mutation the action becomes feasible. Once again this is to ensure exploration of the search space and to provide the search algorithm with the ability to discover new and creative action plans by lowering the constraints across the search space.

3.3.8.4 Crowding Distance.

The crowding distance in NSGAII is utilized to ensure diversity and good spread of points across the Pareto front. Only the least crowded points are accepted into the next population of points. In Jmetal the distance of a solution to neighboring solutions is calculated with respect to each objective function. For a given objective function the two closest neighbors on either side of a solution have their difference taken with respect to the objective function. This distance is then normalized with respect to the maximum and minimum value of the objective function for the current Pareto front. Distance is then aggregated across all objective functions in the manner just presented. The solutions with the higher distance are taken.

3.3.9 Approach to Online Strategic Planning for RTS Agent.

This subsection briefly presents the modified architecture of the agent derived from the original AFIT agent developed in [61]. The main agent module depicted in Figure 3.4 is written in Python and is a multi-scale agent. It is multi-scale in that it has separate managers responsible for specific problem domains of the RTS game as discussed in chapter two. The dotted lines in the figure depict managers that are not yet implemented, however, can be easily supported by the agent framework. Notice that the build manager, strategy manager and unit manager are implemented in the RTS agent.

The build manager is responsible for issuing commands to unary units or units that produce structures or other units. In addition, the build manager monitors state information with respect to volumetric and unary resources. The unit manager is also currently the tactics manager, however, the tactics are scripted and very rudimentary. The unit manager keeps track of a defense group and attack group. Essentially as combat units are constructed they are assigned to the defense group. Once the defense group reaches a certain size specification the defense group units are converted to attack group units and are sent to attack the enemy base. As new units are created they are assigned to the defense group.

This defense to attack group conversion of combat units continues until the enemy or the player is destroyed. The only change to the original AFIT agent is the strategy manager. Previously the strategy manager was scripted to satisfy specific offensive, defensive, and economic parameters for a stipulated strategy - a static strategy manager. With the new agent framework, the strategy manager possesses a planning tool which it utilizes to execute a configured strategy. The strategy manager looks to the CBR mechanism for goals and goal-ordering and then performs a search to determine the build-orders.

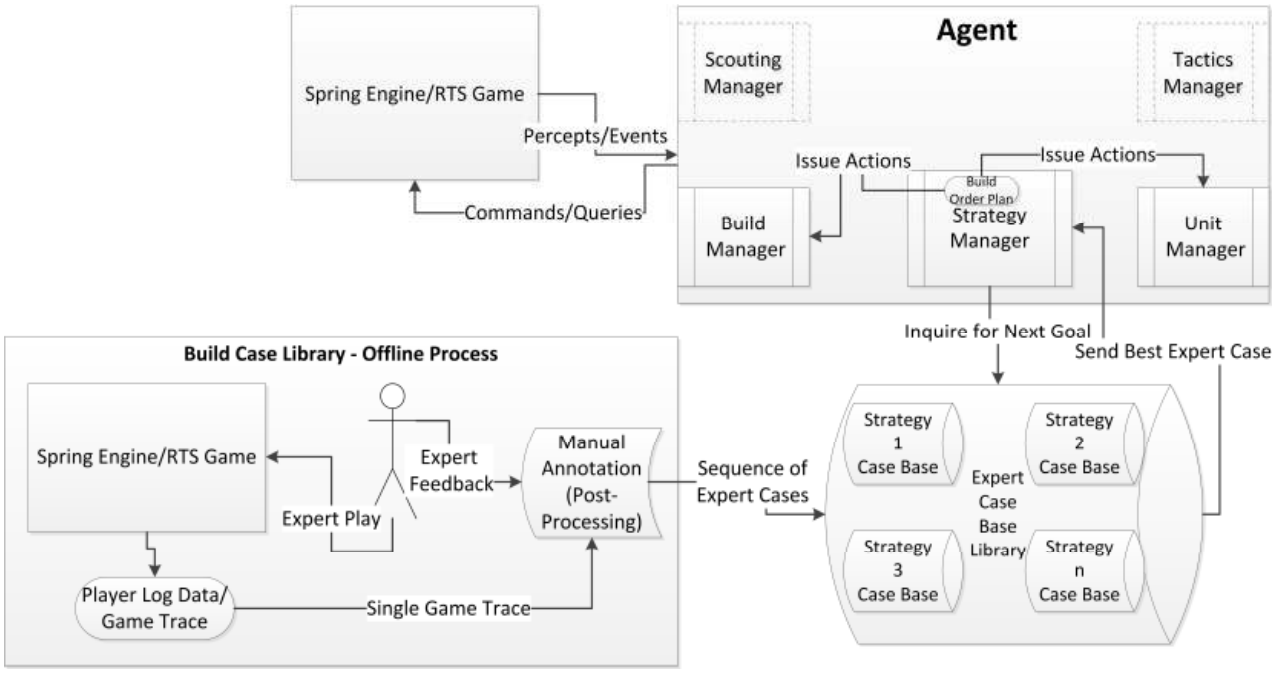

Figure 3.4: AFIT multi-scale agent with CBR for strategic planning.

Figure 3.5 provides a high-level view of how the strategy manager determines which goal to work toward, and how it performs a search to obtain a good build-order plan to satisfy a goal. The strategy manager starts by communicating with the CBR mechanism to determine which goal it should work toward. The CBR contains a database of cases. Unlike other CBR cases such as [65] which contain information on player and opponent states and map properties, the CBR cases we have implemented simply define a desired goal state. This goal state specifies desired unary and combat unit amounts and volumetric resource levels. Cases are organized under a finite number of player strategies. Duplicate cases may exist in the set of strategies included in the CBR database. This simplification of the CBR mechanism lays a foundation for future enhancements. In addition, the current agent architecture lacks a scouting manager, therefore, information on the opponent is limited. Via a case selection function or distance metric, a case or goal is matched with the agent's current game state. The case with the smallest distance is selected for the agent to plan across. The details of the selected case are passed to the planning tool. The planning tool then starts searching for a build-order plan from the agent's current state until a build-order is discovered that brings the agent the closest to the goal state depending on the three objective functions.

Figure 3.5: Agent Strategy Manager

Figure 3.6 is the strategic planning tool. From left-to-right, the strategic planning tool consists of three components: XML schema files describing the RTS game elements and agent game state, the RTS build-order simulator, and the JMetalCpp framework. This planning tool is embedded in the strategy manager of the RTS agent. As the game is played the agent calls the planning tool via an executable generated from the JMetalCpp framework, and only after the CBR mechanism provides it with a goal state to move toward. The JMetalCpp executable is the NSGAII algorithm compiled with specified parameters. The agent communicates to the planning tool via the XML schema files by writing out the current agent game state and goal state to the XML files. Once called by the agent, the NSGAII executable utilizes a user-defined C++ source file that describes a multi-objective problem type called BOO to initialize the Python interpreter and run the RTS simulator. The RTS simulator is responsible for reading in the XML files and executing build-orders provided to it by the NSGAII algorithm. The simulator runs one build-order plan at a time. Once it completes a build-order it returns the fitness score of the build-order and notifies

the BOO problem if it made any modifications to the decision bits of the build-order to ensure feasibility of the action string. This continues until the NSGAII algorithm stops and returns an approximately optimal build-order plan. Essentially NSGAII utilizes the BOO problem to communicate to the RTS simulator. Python is embedded in the BOO problem to call the Python RTS simulator. Note that the final version of the BA simulator utilized by our BA agent is implemented in C++ and accessed via dynamically linked libraries. More on this is discussed in Chapter 4.

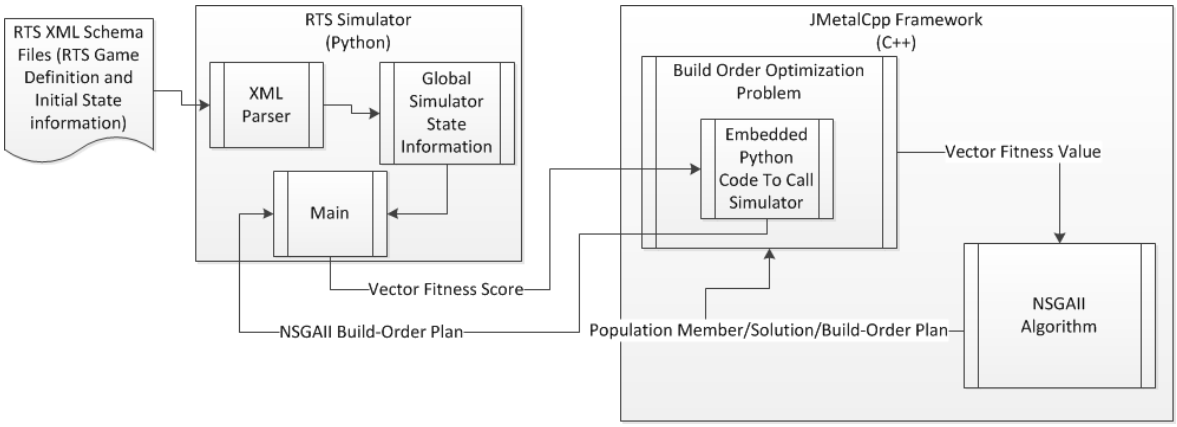

Figure 3.6: Strategic Planning Tool

3.3.10 Universal RTS Build-Order Simulator Architecture.

To simulate build-order executions, we developed an RTS game build-order simulator architecture. The prototype was designed and implemented in Python. It is important to state that the RTS game simulator architecture only simulates build-order decisions it does not simulate tactical decisions (i.e scouting, attacking, moving units). From the architectural design, two RTS simulators were created in order to meet the objectives of this work. The Python language was selected in order to alleviate the burden of implementation, thereby allowing us to focus on design. In addition, utilizing Python allowed us to produce a portable, robust and generic RTS game build-order simulator quickly and easily. The

advantage to utilizing Python, an interpretative language based on C, over higher level languages like C++ or Java, is that it also supports object-oriented development along with a library of easy to use and polymorphic data structures. Python is an excellent prototyping language that enabled us to focus on the design of our mathematical model of the build-order problem as well as the setup of the simulator compared to spending large amounts of time thinking about coding implementation. In general, to write something in Python takes far less time than in C++ or Java and with less lines of code. However, there is a tradeoff in computational time which is addressed in chapter four. We greatly endeavored to develop a RTS build-order simulator that could be easily adapted and utilized by other researchers. How this is possible will become clear shortly. Finally, embedding Python into C or C++ code is fairly straight forward and enabled us to integrate the simulator with the Jmetal MOEA C++ framework. Essentially the tradeoff in selecting Python over C++ was in reducing development time to enable validation and solidification of our design cost of an increase in computational time.

The simulator architecture is designed to be a customizable RTS game strategic decision-making (build-order) simulator. It supports RTS games like Age of Empires (AOE), Starcraft, and Wargus. It is important to note that not all RTS games follow the same formula. For example, the Total Annihilation and BA games are fundamentally different from games like Starcraft and AOE largely in regards to how the economies are structured. This difference required the development of two separate simulators; however, the simulators both respect the MO-BOO model with only subtle differences in implementation. To distinguish these two types of RTS games, games like Starcraft, AOE or Wargus are labeled cumulative economy games and games like Total Annhilation and BA are labeled non-cumulative economy games. Their respective simulators are identified in the same manner. The justification for this naming scheme will become clear in later sub-sections.

The simulator architecture defines an RTS game with seven XML schema files. The schema files fall into three categories: initial game state, goal game state, commands. The initial game state is defined by three xml files. The goal state is defined by three XML files, and the commands category is defined by one XML file. These seven schema files define a single RTS game. Simply put, any RTS game can be simulated by providing its definition in the XML files. We have currently simulated two RTS games with this framework: Starcraft Broodwars and Balanced Annihilation. These games are both RTS games, but have very different game mechanics or strategic approaches. The code for the simulators and their integration into jMetalCpp is available online at [1]. Along with the source code is also included the schema files for Starcraft and Balanced Annihilation.

The simulator architecture is designed around the mathematical formulation of the MO-BOO problem presented in this chapter. It strictly enforces the exist, cumulative, and disjunctive constraints, however, the objective functions can be easily modified to allow users to evaluate various objective functions around the stipulated constraints. The simulator architecture incorporates a fast-forward approach [22][52] in order to evaluate build-order plans quickly.

With respect to implementation, the cumulative and non-cumulative simulators take as input a single action string as defined in our Jmetal solution type. The simulators sequentially assign actions to unary resources. Before being assigned to unary resources, actions must satisfy the exist and cumulative constraints. If these two constraints are satisfied then the simulator attempts to allocate the action to a unary resource which is where the disjunctive constraint must be satisfied. To satisfy this constraint most actions are allocated to a unary resource by waiting until the resource completes the current action it is executing. It is important to stipulate that the RTS simulators only simulate build-orders or macroactions. Tactical decisions like sending out scouts, attacking, moving units,

or damage are not simulated actions. The simulators are strictly defined for simulating build-order decisions or decisions at the strategic level.

3.3.10.1 Handling Actions and Effects with RTS Simulator Architecture.

Similar to planning definition domain language (PDDL) and other planning tools, actions have effects that must be considered after completion of an action. There are three types of effects of actions in RTS games that must be considered: (1) Unary effects, (2) Volumetric effects, and (3) primary effects. Unary and volumetric effects correspond to an action being selected and scheduled for execution. Immediately upon execution there are direct effects on unary and volumetric resources. These include making a unary resource unavailable to other actions (disjunctive constraint) and decrementing the quantity of volumetric resources available. However, these effects are trivial. The important effect is the primary effect of an action. The primary effect is the objective of the action. For example, the primary effect of a 'build_worker' action is to produce an additional worker for the game state. In general, unary and volumetric effects occur immediately at the start of an action, and primary effects occur at the completion of an action. In some games, like BA and Starcraft, actions have more than one primary effect. For example, in BA building a metal extractor has the unary effect of making the commander busy or unavailable, it has the volumetric effect of consuming a calculated production rate of volumetric resources, and two primary effects. The first effect is the increase in collection rate of metal and the second is the increase in the capacity of the agent to store metal. Due to the diversity and in some cases complexity of primary effects the RTS simulator architecture does not strictly define a set of primary effects. Instead these effects are hard-coded into a single function of the simulator that can be modified by a designer as necessary using conditional statements. However, these primary effects are included as attributes for commands defined in the command.xml schema file which translate into variables or constants defined in the simulator code. As the simulator completes an action/command it checks the action's

effects list and processes them as defined in the process effects function of the simulator. The reason for doing this is to allow more flexibility in including and processing effects for actions. An example of why this flexibility is desirable is provided in table 3.7. Also to provide an illustrative example of primary effects the two tables 3.6 and 3.7 present primary effects of BA and Starcraft:

Table 3.6: The table depicts all the primary effects currently programmed into the BA simulator. It introduces the effects by providing a corresponding command that would cause the effect, and what the outcome of the effect will be relative to the command.

Command Name	*Effect Name*	*Description*
Build_ARMlab	Increment	Increases a unary resource's quantity given by the command - in this case the total # of factories.
Build_MetalEx	Accumulate_Metal	Utilized to increase the metal collection rate.
Build_Solar	Accumulate_NRG	Utilized to increase the energy collection rate.
Build_Tank	Increment_Unit	Utilized to increase a combat unit or non-unary unit count.
Assist_Commander	Increment_CmdWorkTime	Utilized to increase the Commander's worktime.
Assist_ARMlab	Increment_FacWorkTime	Utilized to increase a factory's worktime.
Build_MetalStorage	Inc_Metal_Cap	Utilized to increase the metal storage capacity.
Build_SolarStorage	Inc_NRG_Cap	Utilized to increase the energy storage capacity.

Table 3.7: The table depicts all the primary effects currently programmed into the Starcraft simulator. It introduces the effects by providing a corresponding command that would cause the effect, and what the outcome of the effect will be relative to the command. Note the flexibility allotted for defining effects. For example, the simulator treats minerals, gas, and supply as volumetric resources, however, minerals and gas are gatherable resources while supply must be built. Therefore, a different effect is defined for gathering volumetric resources (gas and minerals) and non-gathering volumetric resources (supply). In addition, both gathering actions own two primary effects: Hold and Accumulate.

Command Name	*Effect Name*	*Description*
Gather_Mineral	Hold	Holds a worker to infinitely collect a specified volumetric gathering resource (i.e. minerals and gas).
Gather_Gas	Accumulate	Increments the number of workers collecting a volumetric resources - in this case gas.
Build_Marine	Unit_Increment	Utilized to increase a combat unit or non-unary unit count.
Build_SupplyDepot	Accumulate_Fixed	Increases the total amount of Non-gathering volumetric resources (excludes gas and minerals)
Build_Barracks	Increment	Increases a unary resource's quantity given by the command - in this case the total # of barracks.

3.3.10.2 Advantages and Limitations.

Advantages of our simulator architecture are that you can simulate build-orders for any RTS game that falls into the cumulative or non-cumulative economy categories by defining the game with seven xml schema files. In addition, the simulator enables users to define their own objective functions for scoring build-orders by modifying the Python fitness calculation function defined in the simulator. Since the code is written in Python the simulator is highly portable to any operating system supporting the Python interpreter and can be embedded in most high level languages including C, C++ and Java. A user is provided a framework in which they only need to focus on defining the RTS game they want to simulate. To define a game requires providing definitions in the seven xml

schema files, ensuring the desired effects of actions are captured in a one-to-one matching from the command.xml file to the process effects function of the simulator itself. In addition, the cumulative economy simulator is implicitly implemented with a fast-forward feature to reduce the computation time in scoring build-orders. An important advantage to our simulator architecture is how actions and effects are handled as discussed in section 3.3.10.1. Our approach enables designers to define any strategic level command - strategic level in that it is a build-order command - and its corresponding primary effects. Similar to other simulators from literature [22], our simulator does not take into account map information on locations of resource sites or building locations. We identify this missing variable as the travel time variable. This results in the time required to travel to build sites or volumetric resource gathering locations being left out when producing build-orders. A resolution to this variable would be to provide the simulator with construction unit locations and build site locations in order to account for travel times when determining build-orders. In addition, the simulator assumes infinite volumetric resources. In some RTS games, such as BA, infinite volumetric resources is not an issue, however, for cumulative economic games like Starcraft and AOE, specific mining sites supply a finite amount of a resource. For example, a gold mine in AOE will disappear after a player's workers have extracted the finite amount of gold available from the mine.

3.3.11 Balanced Annihilation Simulator.

As presented earlier, the MO-BOO problem consists of three optimization functions and three constraints. These constraints are the exist, cumulative, and disjunctive constraints. However, unlike cumulative economy games, which must satisfy the cumulative constraint prior to issuing and executing actions, BA is a non-cumulative economy game. In otherwords, issuing and executing an action does not require satisfying the cumulative constraint. In fact this constraint is absent from the MO-BOO model that approximates BA's strategic decision making process. This stems from the fact that BA

utilizes rates of collection and production for assigning volumetric resources to actions. Essentially an action is always feasible with respect to volumetric resources in BA so long as at least one unit is collecting those resources. The rate of collection of a volumetric resource is shared amongst the competing unary resources whose actions' require the volumetric resource. Each action receives a percentage of the rate of collection based on the unary resource's worktime. In BA worktimes define how quickly a particular unary resource can build a structure or unit based upon build duration of the structure or unit being constructed. Build durations are provided by the official BA website [3]. By dividing the worktime of a construction unit into the build duration a build time can be computed in seconds. As stated before this build time in seconds will fluctuate depending on the availability of volumetric resources. If there is a single unary resource executing a single action, a maximum production rate for the unary resource can be calculated given the unary resource's worktime, the duration of the action to be completed, and the current collection rates of the volumetric resources the unary resource requires for production or executing the action. This production rate identifies how much of a volumetric resource a unary resource will utilize per second to complete a target action. As more unary resources simultaneously operate to execute actions, the collection rate of a shared volumetric resource is used to determine how much each unary resource is allocated to use for production. This shared production rate is weighted according to the worktimes of unary resources. A higher worktime guarantees a higher shared production rate versus unary resources with lower worktimes. A unary resources shared production rate will be equal to or less than its maximum production rate depending on the summation of the production rates of all unary resources currently executing and the total collection rate of the shared volumetric resource. The obvious effect of shared production rates is that there is no exact timing for when a unary resource will complete it's action. For example, if a unary resource is provided its maximum production rate then a completion time can be calculated very

simply by dividing the actions duration by the unary resource's worktime. However, if the unary resource is sharing volumetric resources its production rate will fluctuate as other unary resources start and/or complete actions. This means an exact action completion time cannot be determined as can be computed for cumulative economy RTS games like Starcraft and AOE. This makes planning in the non-cumulative RTS game domain more complex than for cumulative economy games since cumulative economy games have known action durations. Known action durations translates into known completion times and start times. As will be discussed later, this led to not being able to efficiently add fast-forward simulation to the non-cumulative RTS game simulator.

This is obviously different from Starcraft or cumulative economies where an action is designated infeasible if the current game state amount of a volumetric resource is not available. In addition, if there is enough volumetric resource to accept the action, the total required amount of the resource is subtracted immediately - there is no sharing or rates of production versus rates of collection. This means an actions completion time is always known. This simplifies fast-forward simulation enabling it to be added to the cumulative economy RTS simulator.

Provided in Table 3.8 and Table 3.9 is a description of the seven schema files and their attributes utilized to define the BA game for the non-cumulative RTS simulator. Though the Starcraft simulator only supports 9 commands at the moment, the BA simulator command XML schema file defines 21 unique commands as derived from the Technology Tree of the BA game.

Table 3.8: BA Simulator Schema Files (XML)

File Name	*State Definition*	*Description*
combat_unit	initial/current	Current number of combat unit or non-unary units agent possesses
unit_goal	Goal	Non-unary resource unit goals (i.e. combat units)
command	N/A	List of RTS game actions agent can take
unary_resource	initial/current	Current number and type of unary resources in existence
unary_goal	Goal	Unary resource goals
vol_resource	initial/current	Current quantities of volumetric resources
vol_goal	Goal	Volumetric resource goals

Table 3.9: Attributes (Attr) of the seven XML schema files. MC: Metal Cost; EC: Energy Cost; CR: Collection Rate; PR: Production Rate; Res: Resource; Vol: Volumetric; Act: Action

XML File Name	**# Attr**	**Attr**					
combat_unit	2	Unit Type	Current Amount				
unit_goal	5	Unit Type	Time to Build	Target Amount	MC per Unit	EC per Unit	
command	6	Act Name	Duration	Vol Res List	Unary Res List	Exist List	Effects List
unary_resource	2	Unit Type	Current Amount				
unary_goal	5	Unit Type	Time to Build	Target Amount	MC per Unit	EC per Unit	
vol_resource	6	Res Type	Capacity	Current Amount	CR	PR	
vol_goal	3	Res Name	Collection Rate	Initial Amount			

IV. Design of Experiments

4.1 Introduction

This chapter introduces the objectives of our experiments and their importance. A discussion on parameter settings for the NSGAII algorithm and our planning tool is also presented. The objectives of our experiments are as follows:

1. **The first experimental objective of this work is to validate that the mathematical model of the build-order problem introduced in Chapter 3 is capable of planning in the RTS domain. This is demonstrated across planning goals for Starcraft: Broodwars and Balanced Annihilation.**

2. **The second experimental objective is to demonstrate the capabilities of the planning tool across varying parameter settings. This is conceptualized via a Pareto front.**

3. **The third experimental objective is to demonstrate that our strategic planning tool, utilizing the mathematical model, can be used as an online planning tool by an RTS agent to play a full RTS game at or near an expert level.**

4. **The fourth experimental objective is to validate a method for determining player skill-level in an RTS game. This objective is achieved in connection with the first three experimental objectives.**

As stated in Chapter 3, the Python language is selected for developing the RTS game simulator in order to develop a simulator that can be easily modified and run. Through fast modifications and test runs we are able to ensure valid design of our build-order MOP and RTS simulations for the various RTS games. Another key goal was to provide source

code that can be easily understood and adapted by other researchers interested in the MO-BOO design. As anticipated, the trade-off to this flexibility and shorter development time is slower computational time in planning, compared to a C++ or Java implementation of the RTS simulator. With a valid and functional Python prototype, the strategic simulator is translated into C++. This translation reduces planning time by greater than 85%. This also enables moving the planner into the online phase of our development process, which entails integrating the planning tool with a pre-existing agent developed in [61] for BA.

Recall that the simulator does not take into account two variables: travel time and finite volumetric resources. Since the simulator does not take into account map information when determining build-orders, this results in build-orders that do not account for travel time of construction units to build-sites. The second variable is that the simulator is designed under the assumption that volumetric resource sites possess an infinite amount of resources. This second variable is only an issue for Starcraft, in BA volumetric sites are in fact infinite. To overcome the first variable in BA, we position all agents and opponents in a starting location so that they are as close to resource sites as possible. This keeps the game fair and also minimizes the affects of travel time in the overall execution time of the strategy provided by the strategic planning tool to agent Boo. To remove this variable would involve providing the simulator with map information in terms of build-site locations and construction unit locations.

4.2 Experimental Setup

4.2.1 Machine and Software Specifications.

For experimentation we utilized the Ubuntu 11.10 32-bit operating system, Spring Engine V91.0, Balanced Annihilation V7.72, Starcraft: BroodWars, Python 2.7, JMetalCPP v1.0.1. All C and C++ code can be compiled with GCC 4.6.1 and Intel's compiler ICC. The hardware specifications are 8GB Memory, 2.80GHz Intel Core2 Duo, and NVIDIA Quadro. Matlab 2013a was utilized for analysis.

4.2.2 Balanced Annihilation Agents.

This section introduces the three agents used for conducting our experiments. Agents LJD and BOO were both developed at AFIT to play the BA game. Agent E323 comes as a standard AI for the Spring Engine BA game.

4.2.2.1 Agent LJD.

Agent LJD is the BA multi-scale AI developed at AFIT for the Spring Engine. We consider agent LJD to behave at an expert level with regards to strategy execution because the agent's strategy manager is scripted by an expert player. More on this agent and its design can be found in [61]. Agent LJD begins a game session by establishing an economy which consists of building metal extractors for gathering the volumetric resource metal, and solar panels for gathering the volumetric resource energy. Once volumetric resources are being gathered, the agent then builds a factory (unary resource) capable of producing combat units. In BA there are three types of factories: k-bot lab, vehicle plant, and aircraft plant. A vehicle plant produces tanks and other wheeled or tracked combat vehicles, and the k-bot lab produces mechanized assault bots. It is important to note that agent LJD never advances to a higher technology level. In BA there are two technology levels: basic and advance. Each unit factory requires the basic version to be produced first before advancement to the second technology level. Agent LJD always remains at technology level one because this is how the agent was scripted in [61]. The quantity and names of the respective combat units produced under agent LJD's strategies are outlined in Table 4.17. Agent LJD continues to produce the same exact combat units specified in the table. With completion of a set of combat units a new iteration begins. This iteration of unit production is known as attack waves.

Agent LJD implements a very simple tactical process. As combat units are produced they are assigned to defend the commander. As soon as the first wave of troops are complete, this wave then becomes an attack group and marches toward the enemy base.

As more units are produced they are assigned to defend the commander. This cycles goes on until the game is won or loss.

4.2.2.2 Agent BOO.

Agent BOO - Build-Order Optimization - is our newly implemented agent. This agent utilizes our strategic planning tool to discover and execute build-orders that are as good as or better than expert build-orders. For experimental purposes only, Agent BOO executes any one of the strategies outlined in Table 4.17. In an offline process, the strategies are manually broken down into subgoals and placed into the CBR case dataset. Once assigned a strategy, agent BOO's planner selects a case from that specified strategies case set. The planner selects a case based upon agent BOO's current game state and passes the goals of the case to the planning tool. Throughout experimentation Agent BOO's planning window is limited to no more than fifteen actions in order to minimize planning times and enforce intermediate goal planning [19]. More on this is presented in section 4.4.2. A planning window as defined in [63] is the maximum number of actions an agent can execute to reach a goal state. Figure 4.1 presents the technology tree of BA that agent BOO is able to traverse. Once more, agent BOO inherits the technology level restriction of agent LJD. With the current BA simulator implementation, agent BOO is capable of planning across 21 unique decisions. These commands are derived from the technology tree presented in 4.1 as well as from agent LJD. They enable agent BOO to plan across the entire decision space captured in the technology tree. The planner can be modified to incorporate more BA RTS actions via the *Command.xml* schema file, however, it was unnecessary for our experiments since agent BOO is limited to the first technology level.

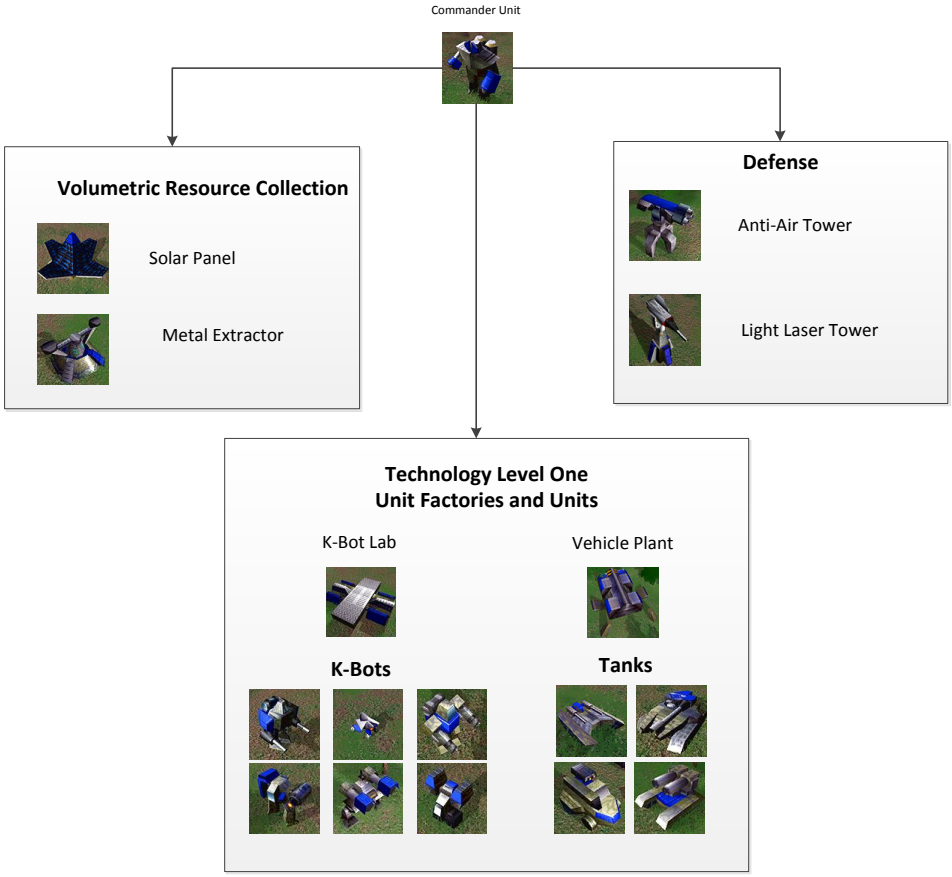

Figure 4.1: The level one technology tree from the BA game that agent BOO is able to traverse.

4.2.2.3 Agent E323.

The Spring RTS engine provides several built-in AI opponents. For this work we selected agent E323 to play against BOO in order to observe two things: how agent BOO responds to an opposing force, and how adaptable agent BOO is to losing units and infrastructure. Agent E323 utilizes a combination of two strategies: k-bot rush and blitz. It first sends an initial wave of k-bots to hinder enemy infrastructure - slow down resource consumption - and then begins building a larger attack force to send out in waves, and

builds light defenses around its home base. It is important to clarify that we assume agent E323 is a scripted agent and is not designed around the multi-scale agent framework. This is unlike agent LJD and agent BOO. Agent LJD is a multi-scale agent with its managers' behaviors scripted by an expert. Whereas agent BOO is a derivation of agent LJD with all managers scripted except for the strategic manager which utilizes our strategic planning tool to generate build-orders and select goals.

4.2.3 Metrics.

The DBD metric complements the third objective function of our build-order MOP - makespan. It is expected that a player with a lower DBD produces build-orders with smaller makespans than players with higher DBD values. This argument is derived from our conjecture that an expert player executes strategies faster than non-expert players. A strategy is executed as an explicitly defined build-order across the duration of the game or the concatenation of all build-orders executed throughout the game. We utilize the DBD metric to quantify agent BOO and agent LJD in order to provide more perspective on how the two agents compare relative to strategy implementation or skill level in executing a strategy.

4.2.3.1 Measuring Player Skill.

In Chapter 3 the arithmetic DBD measure is first introduced. After further evaluation, we formulated a second version of the DBD metric which we call the geometric DBD. The following expression is geometric DBD (ensuring to remove feature values of zero from the computation):

$$\frac{\prod_{i=1}^{E}(D_{i+1} - D_i) * 1000 * N}{(E - N)} \tag{4.1}$$

E is the total number of features or decisions in the player vector and N is the total number of times the player does not make a decision - in a player vector this is expressed as the value zero. The geometric mean maintains the consistency of the data

over the entire time period the player is making decisions, whereas the arithmetic mean is susceptible to infrequent decisions times that may exceed the median decision time. This is an important to distinction because both metrics can be utilized to understand the skill level of RTS players. The arithmetic DBD provides a single value that states clearly which expert completed the strategy first. On the other hand, the geometric DBD captures the consistency between milestone decisions in a players strategy execution. This distinction is clearer in 4.5.3.5. Once again we provide the plots of the non-filtered and filtered Starcraft expert player data first presented in Chapter 3, where filtered means we removed players from the data that did not satisfy a decision threshold of at least 20 decisions. We assume that players who make fewer than 20 decisions over the entire feature set of 50 possible decisions are not experts but outliers in the dataset.

From figure 4.3 it is clear that experts across the three strategies operate within a distinct geometric DBD range between 1.4 to 2.4. Notice that the arithmetic means of the geometric DBD for the filtered expert strategies are all close to 1.8 and their corrected sample standard deviations are close to 0.2. This data reveals a consistency in expert strategy execution time regardless of the strategy being executed. Figure 4.2 is depicts the non-filtered dataset of strategies.

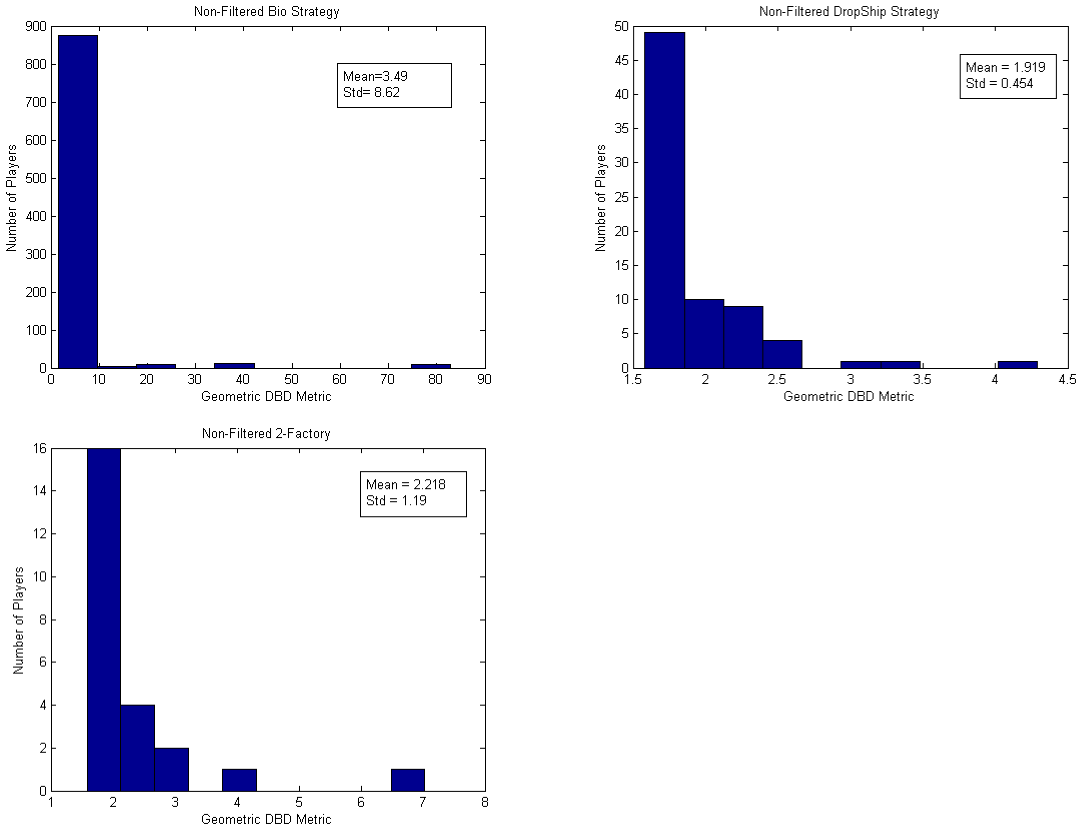

Figure 4.2: Distribution of the geometric DBD metric for expert players of Starcraft: Broodwars. This data is prior to filtering for outliers.

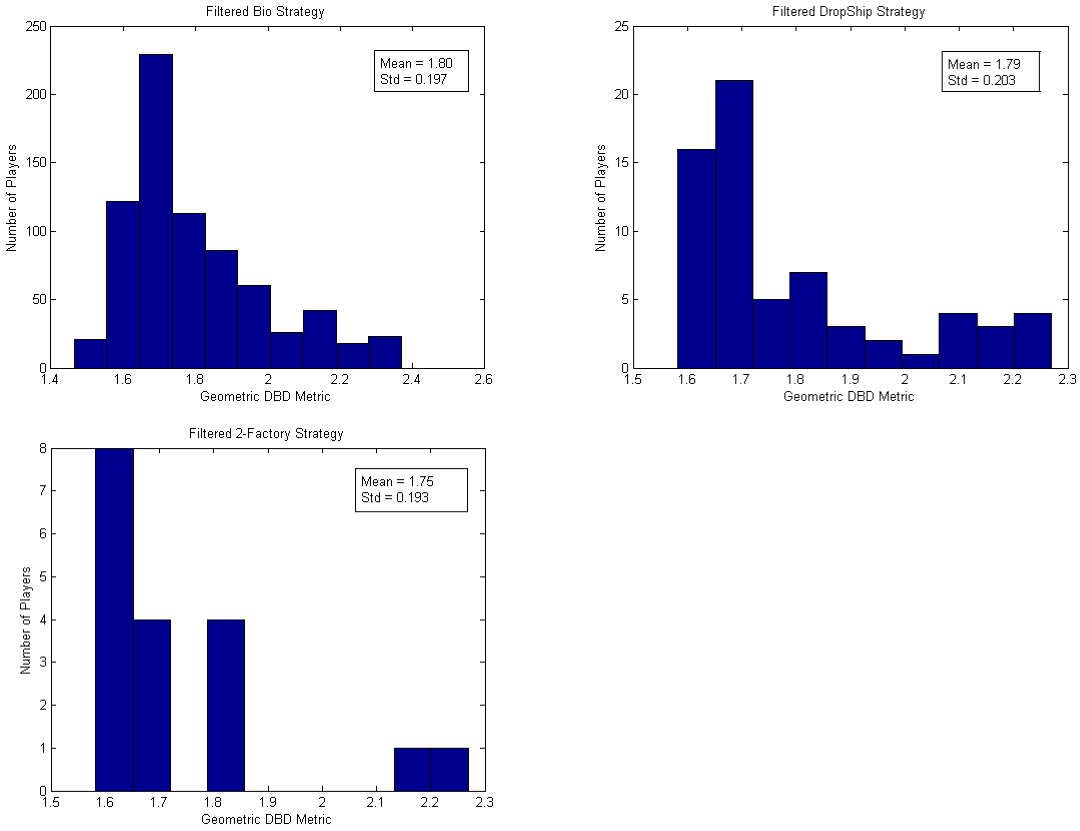

Figure 4.3: Filtered for outliers, these histograms depict the distribution of the geometric DBD metric for expert players of Starcraft: Broodwars.

4.3 Experimental Objective One

To achieve experimental objective one we have designed the following experiment. We utilize the Starcraft strategic planner in an offline setting and provide initial build-order goals from the Starcraft: Broodwars game. Results on BA are presented in later sections. The planning tool then outputs the strategy along with its fitness values for the three objective functions ($objective1, objective2, objective3$).

4.3.1 Experimental Objective One Results and Analysis.

We utilized the following parameters displayed in Tables 4.1 and 4.2 for the Python implementation of the Starcraft strategic planner. Recall that the Python implementation is only suitable for offline planning due to the large computational time.

Table 4.1: Starcraft Python Planner: NSGAII parameters and average execution time - execution time is not a parameter, but the time to determine a build-order.

Population Size	**Num Evaluations**	**Mutation Rate**	**Crossover Rate**	**Bit Length**	**Offline Execution Time (secs)**
50	6000	1/k	0.9	k=60	37.69

Table 4.2: Starcraft Python Planner: Simulator Parameters

Mineral Collect Rate	**Gas Collect Rate**
1.35/sec per worker	2.1/sec per worker

The reason for using the parameter settings described in Table 4.1 is because we empirically found these to be satisfying parameter values for the BA online planner written in C++ - this is discussed more in Sections 4.4 and 4.5. We conjecture that if we translated the Starcraft Python simulator into C++ code we could effectively reduce the computational time so that the Starcraft planner could be utilized as an online planner just as the BA C++ implementation is utilized by our BA agent in Section 4.5. Also note, that a Starcraft C++ version would most likely run faster than our current BA C++ implementation due to the fact that the Starcraft build-order simulator supports fast-forward simulation [22], whereas, the BA C++ simulator does not. Integrating our planner into a Starcraft agent is not an objective of our research. We simply include the discussion to demonstrate that our mathematical model or build-order MOP and design is capable of solving the build-order problem for various RTS games.

The planner is first provided an initial game state in Table 4.3 and a goal state in Table 4.4 defined via XML schema files. By running the planner over four iterations and injecting the best solution found over each previous iteration into the new population of the NSGAII algorithm, we obtained the build-order presented in Table 4.6. The final state reached is presented in Table 4.5. We also observed that by increasing the evaluations to twenty-four thousand our planner found the same build-order in one run; however, twenty-four thousand evaluations is not suitable for online planning due to computation time. Our planning tool is able to determine near-optimal build-orders without prior build-order plans from an expert strategy and minimal parameter settings. Bounding the search using expert domain knowledge is unnecessary, unlike global search techniques such as DFS and BFS [22][19].

Table 4.3: Starcraft: Initial State

Resource	**Amount**
Mineral	100
Gas	0
Cmd Ctr	1
Barracks	0
Supply	5
Marine	0
Worker	5
Refinery	0

Table 4.4: Starcraft: Goal Definition

Resource	**Amount**
Mineral	-
Gas	-
Cmd Ctr	-
Barracks	-
Supply	-
Marine	7
Worker	-
Refinery	-

Table 4.5: Game state reached by planner offline.

Resource	**Amount**
Mineral	122
Gas	0
Cmd Ctr	1
Barracks	2
Supply	7
Marine	7
Worker	6
Refinery	0

The build-order generated from the planner is presented in Table 4.6. The build-order starts by assigning three of the five workers from the initial game state to gathering minerals. It then assigns the remaining two workers to construct two barracks and a supply depot. While the construction is occurring, marines are trained at the barracks as they complete, and the command center (Cmd Ctr) is then tasked to build an additional worker. Lastly, a fourth worker is tasked to mine minerals and a final marine is trained. The score of this build-order (0,0,192) signifies that the best time to train seven marines is 192 seconds if the initial game state is as described in Table 4.3. However, the strategic planning tool is intended to receive goals and goal-ordering from an expert strategy utilizing a case-based reasoning mechanism. This example is a toy demonstration of the planners capability to reach a goal state as quickly as possible from an initial state without the assistance of an expert build-order. In our Starcraft simulator, once a worker is assigned to mine minerals or collect gas they can never be reassigned for another task. Also workers utilized for construction can be assigned to any construction project or resource gathering action; however, once assigned for resource gathering they cannot be released.

Table 4.6: Build-order Plan Returned. Collect Mineral (M), Build_Barracks(B), Build_Supply(Sp) Train Marine(TM), Train Worker (W). The fitness score is (0,0,192).

M	M	M	B	B	Sp	TM	TM	W	TM	TM	TM	TM	M	TM

4.4 Experimental Objective Two

This section details our design decisions for modeling the build-order problem as a MOP and the significance of the genotype utilized by our MOEA. The focus of this section is to present the capabilities of the planning tool prior to its application in the RTS Spring Engine with agent BOO.

4.4.1 Pareto Front.

As presented in chapter three the build-order MOP or MO-BOO problem is defined by three objective functions and three constraints:

$$min(\sum_{i \epsilon A_G}^{|A_G|} D_i * (\{G_i\} - \{S_i\})) \tag{4.2}$$

$$min(\frac{1}{C_r}[\sum_{r \epsilon A_G}(A_{G_r}) - R_{S_r}]) \tag{4.3}$$

$$min(\sum_{i \epsilon A}^{N} F_i) \tag{4.4}$$

constraints,

$$B \implies A \tag{4.5}$$

$$\sum_{a_j \in A}^{|A|} r_j^t \leq L_t \tag{4.6}$$

$$end\_time(A) \leq start\_time(B) \vee end\_time(B) \leq start\_time(A) \tag{4.7}$$

MO-BOO Objective Functions

1. The first objective function 4.2 is to minimize the duration of actions from the set A_G defined by the goal state not taken by the agent in its current game state S_i.

2. The second objective function 4.3 is to minimize the difference in required volumetric resources (defined by the goal state and its action set) and available volumetric resources in the agent's current state for all volumetric resources required by the actions in A_G, or said in another, way to minimize the time required to collect the volumetric resources needed to execute the set of actions in the goal state or A_G.

3. The third objective function is makespan. The objective is to minimize the time required to execute a set of actions A.

MO-BOO Constraints

1. The exist constraint, 4.5, is a implication statement where B and A are actions. This expression draws a logical implication between actions B and A. It captures the relationship that building an armory to provide weapons to soldiers is unnecessary if there is no barracks to train soldiers first. In this case there are two unary resources, an armory and a barracks. The barracks produces combat units, and the armory provides upgrades to weapons and armor to enhance the combat effectiveness of the soldiers. In this particular situation, the exist constraint enforces that any action to build an armory can only occur after a barracks is constructed or in existence. Note that the action to build an armory does not utilize the unary resource of barracks to perform any action, merely that it exist. The unary resource which must exist for an action to occur will not also be the unary resource utilized by the action to execute. For example, a command center in Starcraft is used to build workers. This means the command center is a unary resource used by the action *build_worker*. Though the command center must exist in order for the action to be assigned to the command center, this is not what is meant by the exist constraint. This constraint is stipulated by the technology tree of an RTS game and requires that a unary resource must exist in the agent's current game state, but is not assigned work by the action requiring the resource to exist.

2. Once more the cumulative constraint defined by equation 4.6 stipulates that given a set of action $|A|$ the agent can only execute actions within this set if the supply of volumetric resources of the agent's current game state, L_t, are large enough to support the execution of the actions in set $|A|$. This constraint pertains to actions requiring

sequential and concurrent execution and volumetric resources. In most RTS games every action requires two unique volumetric resources.

3. The disjunctive constraint applies only to actions requiring unary resources, which at least every action in most RTS games requires at least one unary resource. This constraint requires that any two actions competing for a single unary resource must satisfy a domain of times where an action B either starts after the end time of action A or action A starts after the end time of action B. Essentially no two actions can be executed simultaneously on any one unary resource. In order to execute concurrently multiple copies of the unary resource must exist.

The first two objectives provide a path to the goal, however, this path is not required to be optimal with respect to time. With the addition of the third objective, the path now becomes optimal in terms of makespan. It is obvious that the first objective is intended to ensure actions defined in the goal state are taken. The second objective can be viewed in two ways depending on how a goal state is defined. In one way it is intended to ensure resources are collected to satisfy the execution of the set of actions stipulated in objective one, but if a goal is also defined in terms of volumetric resource levels, objective two also ensures collection of those resources. Finally the third objective ensures that actions are scheduled and executed to minimize time. Together these functions define a tradeoff surface with respect to actions, resources, and time.

To visualize the tradeoff surface of our three objective functions we plot several Pareto fronts produced by our strategic planner tool for the initial BA state and goals defined in Table 4.7.

Table 4.7: Initial State and formulated Goals.

Resources	Initial State	Small Goal	Large Goal
armvp	0	-	-
armmex	0	-	-
armsolar	0	-	-
armstumpy (tanks)	0	3	6
metal C.R	1.5	-	-
metal Amt	1000	-	-
energy C.R	25	-	-
energy Amt	1000	-	-

To produce the Pareto fronts, the strategic planning tool settings in Tables 4.8 and 4.9 are used because they have empirically demonstrated to provide computationally fast and near-optimal results with respect to various BA strategies. These parameters tune the MOEA and simulator so that the planner is suitable for online use by an RTS agent in the BA game. The *BitLength* parameter is the action string length or genotype size of the solutions. This relates to the maximum number of actions the planner can return for a build-order. In most cases, the planner returns build-orders with a fraction of the actions represented by the bit string.

Table 4.8: Balanced Annihilation C++ Planner: NSGAII Parameters and planning tool execution time - execution time is not a parameter, but a performance metric.

Goal	Population Size	Num Evals	Mutation Rate	Crossover Rate	Bit Length	Runs	Execution Time (secs)
Small Goal	50	6000	1/k	0.9	k=60	5	6.09
Small Goal	50	6000	1/k	0.9	k=100	5	6.70
Small Goal	50	6000	1/k	0.9	k=200	5	14.32
Large Goal	50	6000	1/k	0.9	k=60	5	8.778
Large Goal	50	6000	1/k	0.9	k=100	5	9.92
Large Goal	50	6000	1/k	0.9	k=200	5	15.75

Table 4.9: Balanced Annihilation C++ Planner: Simulator Parameters

Metal Collect Rate	Energy Collect Rate
2.04/sec per metal extract	20/sec per solar panel

The Pareto fronts computed are depicted in Figures 4.4 and 4.5. These fronts reveal that the larger the action string or genotype length of the solution the closer each objective function is to zero. This provides a decision surface that brings an agent closer to the goal. For a decision maker, RTS agent or expert designer, the best solution from the solutions presented in the Pareto fronts might be the one that minimizes objectives one and two to zero and has the smallest constant value for objective three. In the Pareto fronts presented, objective three is measured on the vertical axis; therefore, a point in objective space lying on this axis reaches the goal, but may not be optimal with respect to makespan. Again this is a tradeoff surface. Another, decision maker may decide that the solution that is close to the goal, but executes faster than a solution lying on the vertical axis is a better choice. For example $(10, 0, 100)$ may be considered a better solution then $(0, 0, 300)$ since only ten additional seconds are required to reach the goal moving the actual makespan to 110 seconds over 300 seconds.

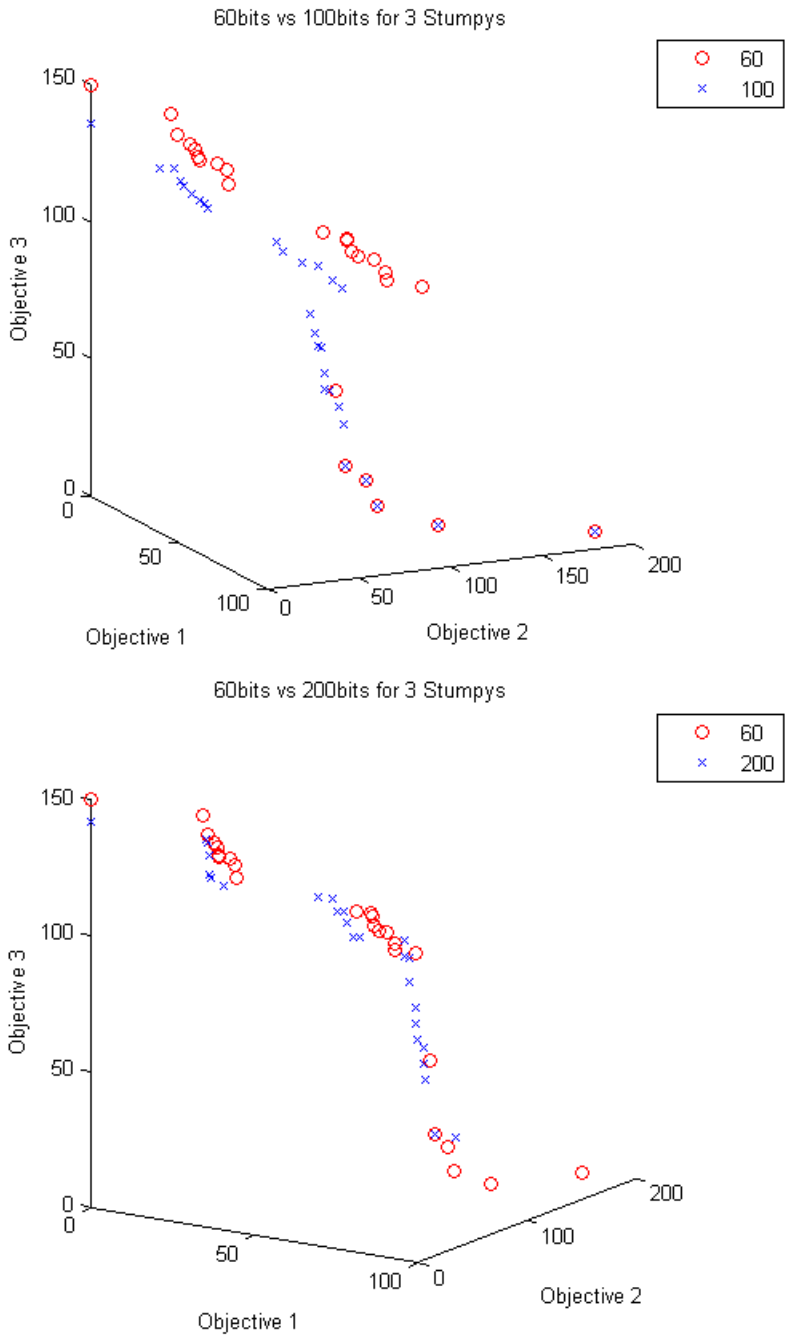

Figure 4.4: Comparison of the Pareto fronts produced by the parameters in table 4.8 for the small goal requiring 12 actions.

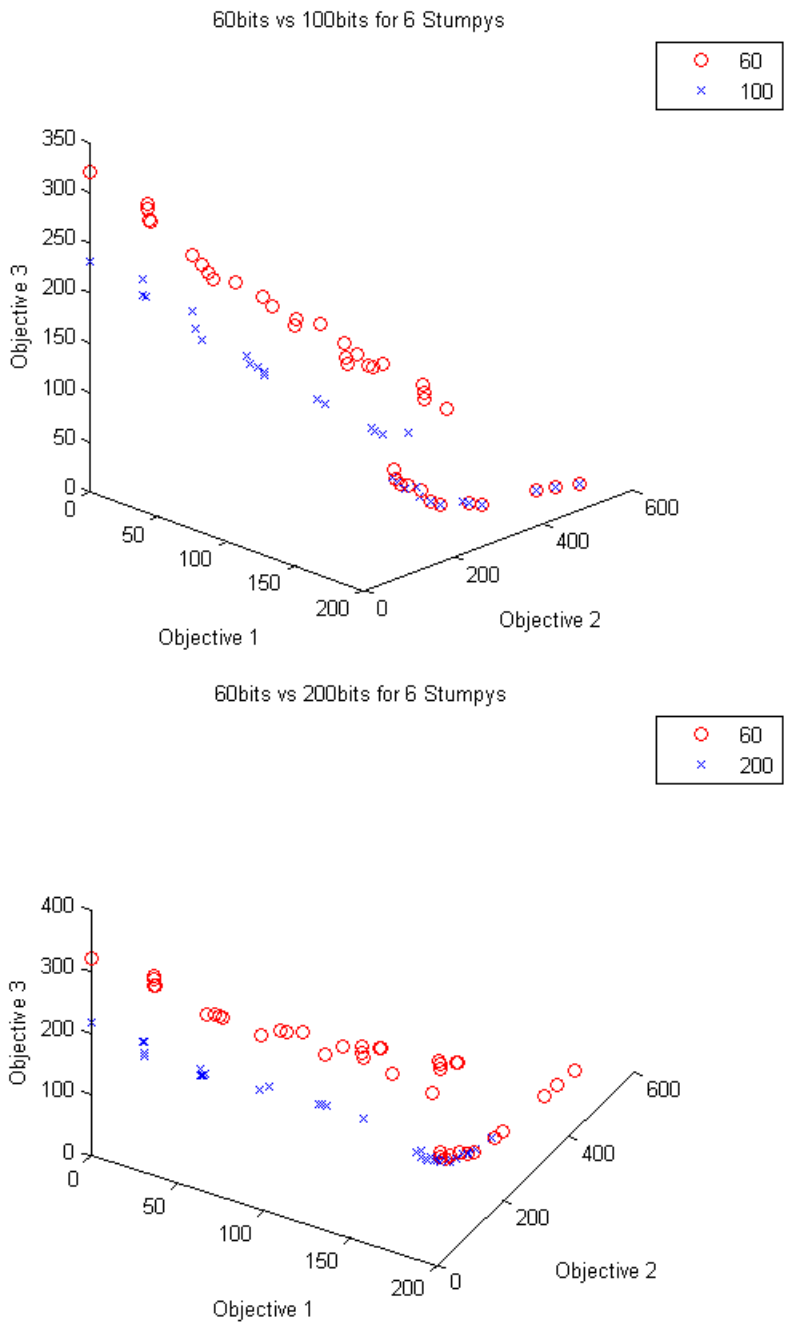

Figure 4.5: Comparison of the Pareto fronts produced by the parameters in Table 4.8 for the small goal requiring requiring 24 actions.

The best build-orders with respect to makespan and reaching the goal state from the Pareto fronts are presented in Table 4.10. These results reveal that the larger the genotype the better the results, but the larger the computational time as noted in Table 4.8. The results are from the best run of a set of five runs. Each result is calculated without injecting expert build-orders into the population of the MOEA. The results presented are to demonstrate the abilities of our build-order MOP and strategic planning tool, and are not necessarily the best methods for reaching a goal. Table 4.10 shows that a good build-order plan for the *SmallGoal* is 12 actions long and can be completed in simulation in 136 seconds. With respect to the *LargeGoal*, the goal can be reached in 24 actions with a time of 198 seconds to execute in the simulator. These action sizes reveal the planning window size each bit length is able to perform well in. Empirically we have found that the 60bit planner operates well with a online planning window size of 12-15 actions- note that this is without the aid of expert build-order injection into the NSGAII population and an evaluation constraint of six-thousand for online performance.

Table 4.10: Balanced Annihilation C++ Planner: Build-Order Results. The best build-orders are marked with $*$.

Goal	Bit Length	Fitness Score	Planning Window
Small Goal	k=60	(0,0,150)	-
Small Goal	k=100	*(0,0,136)	12
Small Goal	k=200	(0,0,142)	-
Large Goal	k=60	(0,0,322)	-
Large Goal	k=100	(0,0,232)	-
Large Goal	k=200	*(0,0,198)	24

4.4.2 Intermediate Planning vs Singular Planning .

In [19] the authors observe that planning improves or makespan decreases with the use of intermediate goals versus a larger singular goal. Our observations utilizing our

planning tool in BA support this finding. We first extract two planning goals from the strategy executed by LJD for the BA Turtle strategy. This is achieved by examining LJD's player vector for actions and times. We then evaluate the build-order time produced by our planning tool offline to reach goal two by first reaching goal one- intermediate planning. We then utilize our planning tool to observe the time to reach the second goal without the use of the first goal - singular planning. We observed that the agent reaches goal two faster via intermediate planning. This points out the significance to how important goal ordering is when planning in the RTS domain. For our experimentation we assume that the experts we derive our goals from execute strategies with near-optimal goal ordering.

The results of this experiment in intermediate planning versus singular planning are presented in Tables 4.11, 4.12. The results are important because they demonstrate that producing expert level build-orders across properly ordered subgoals can result in expert level execution of a larger goal. Therefore, near-optimal execution of a strategy is preserved across well-ordered subgoals. Once again, we assume a near-optimal ordering of subgoals can be derived from expert strategies and provided to our planner via CBR which we demonstrate in objective three of our experiments.

Table 4.11: Agent BOO - (SIM only) Intermediate Planning

Resources	Initial State	Goal One	Goal Two
armvp	0	1	1
armmex	0	3	5
armsolar	0	4	4
armflash	0	0	4
armsam	0	0	2
armllt	0	0	2
armrl	0	0	2
metal C.R	1.5	7.62	11.7
metal Amt	1000	5.1746	26.117
energy C.R	25	105	105
energy Amt	1000	1200.00	43.166
Game Time	0	**82**	**188**

Table 4.12: Agent BOO - (SIM only) Singular Planning

Resources	Initial State	Goal Two
armvp	0	1
armmex	0	3
armsolar	0	3
armflash	0	4
armsam	0	2
armllt	0	2
armrl	0	2
metal C.R	1.5	7.62
metal Amt	1000	0.276
energy C.R	25	85
energy Amt	1000	235.799
Game Time	0	**218**

To further validate our conjecture that intermediate planning is better over singular planning when a good ordering of subgoals is available, we execute the planner in game

via our BOO agent. The results are presented in tables 4.13 and 4.14. Observe that the in game time to reach goal two only varies by nine seconds between the two planning techniques. This is due to the travel time variable of the BA simulator as well as the fact that the intermediate planning technique produces an additional *armsolar* building. Also note that their are subtle energy collection rate differences between the simulator game states and real-time game states. This is because the simulator does not account for smaller energy or metal rate collection increases introduced by additional units. It only factors in metal extractors and solar panels, which are responsible for the majority of volumetric resource collection.

Table 4.13: Agent BOO - (In-Game) Intermediate Planning

Resources	Initial State	Goal One	Goal Two
armvp	0	1	0
armmex	0	3	0
armsolar	0	4	0
armflash	0	0	4
armsam	0	0	2
armllt	0	0	2
armrl	0	0	2
metal C.R	1.5	7.614	9.653
metal Amt	1000	192.388	58.85
energy C.R	25	105	107.5
energy Amt	1000	1200	28.23
Game Time	0	**133**	**241**

Table 4.14: Agent BOO - (In-Game) Singular Planning

Resources	Initial State	Goal Two
armvp	0	1
armmex	0	3
armsolar	0	3
armflash	0	4
armsam	0	2
armllt	0	2
armrl	0	2
metal C.R	1.5	7.614
metal Amt	1000	0.305
energy C.R	25	87.5
energy Amt	1000	289.55
Game Time	0	**249**

In order to demonstrate the optimality of the build-order produced by Agent BOO via our strategic planning tool, we introduce the goal completion times and final goal states achieved by Agent LJD via simulation in table 4.15 and in-game in table 4.16.

Table 4.15: Agent John - (SIM only) Intermediate Planning

Resources	Initial State	Goal One	Goal Two
armvp	0	1	1
armmex	0	3	5
armsolar	0	4	4
armflash	0	0	4
armsam	0	0	2
armllt	0	0	2
armrl	0	0	2
metal C.R	1.5	7.62	4.566
metal Amt	1000	8.1469	1.146
energy C.R	25	105	105
energy Amt	1000	1200	1200
Game Time	0	**85**	**227**

Table 4.16: Agent John - (In-Game) Intermediate Planning

Resources	Initial State	Goal One	Goal Two
armvp	0	1	0
armmex	0	3	0
armsolar	0	4	0
armflash	0	0	4
armsam	0	0	2
armllt	0	0	2
armrl	0	0	2
metal C.R	1.5	n/a	n/a
metal Amt	1000	n/a	n/a
energy C.R	25	n/a	n/a
energy Amt	1000	n/a	n/a
Game Time	0	**133**	**276**

4.4.3 Summary of Intermediate vs Singular Planning.

Section 4.4.2 provides insight into how our strategic planning tool can be utilized by an RTS agent to optimize build-orders derived from expert strategies. Once again, the optimality of a build-order produced by the planner relies on the number of actions required to reach a goal, the action string length of the genotype, the strategy selected for execution, and the ordering of subgoals derived from an expert strategy. In addition, we provide an evaluation of the capabilities of our build-order MOP and strategic planning technique to be utilized across the RTS genre - specifically for cumulative games like Starcraft and non-cumulative games like BA.

4.5 Experimental Objective Three

The third experimental objective is to validate and emphasize the online characteristic of our strategic planning tool as well as show how the planner provides an agent with the means to overcome changes to the game state. In order to demonstrate the achievement of this objective, we utilize the Spring RTS game engine. Building upon some of the work

in [61], we modified agent LJD to incorporate our online strategic planning tool, thereby creating agent BOO. To demonstrate the planning capability of our planning tool we place agent BOO against a NULL AI in the BA game. NULL AI is the name of an agent in the Spring game that does nothing. The use of this agent allows agent LJD and agent BOO to plan and execute their strategies in an environment where they are not in danger of being under attack. The purpose of this is to constrain the game environment in order to minimize outside influences on the agents' build-orders to enable strategy execution comparisons. Utilizing agent LJD as an expert player, in regards to strategy execution, we establish a baseline to evaluate the performance of the strategic planning tool utilized by our agent BOO.

We hypothesize that agent BOO should execute strategies at a more advanced player skill level than agent LJD, as well as demonstrate optimality with respect to overall strategy execution time. To make these assessments the actions of the agents are time stamped and outputted to text files. These text files are converted into player vectors which follow a strategy schema similar to the technique utilized by Weber [65]. Several of the BA strategies defined in [61] are formatted as strategy schemas as Weber did. We compare when and how much faster the agents execute actions to reach subgoals and the overall time to execute the entire strategy. Again to measure skill level we apply the arithmetic and geometric DBD functions to the player vectors.

As part of experimental objective three, we also want to demonstrate that agent BOO adapts to a changing game state. We place BOO against agent E323 and observe how agent BOO responds after assaults from enemy units to its home base - the CBR mechanism with the planner should enable agent BOO to adapt. This experiment involves placing agent BOO against a more advanced Spring Agent,relative to agent NULL AI, and observing how agent BOO selects goals from the CBR system as it loses units and infrastructure to combat and attacks.

4.5.1 Evaluated Strategies.

In [61] the author describes eight strategies that can be utilized in the BA RTS game. From those eight strategies we selected three to evaluate agent BOO. The strategies are outlined in table 4.17. These strategies are selected because they provide more decision making opportunities for agent BOO to plan across. From these three strategies and observing the actions of agent LJD, a set of goals are defined. These goals respect the ordering of subgoals captured by agent LJD's player vector. The goals for each strategy are translated into cases that define our CBR mechanism utilized by BOO. The CBR mechanism is a large XML file consisting of cases for a single strategy only. As BOO plans, the agent selects a goal from the CBR system based upon its current game state. It then publishes a selected goal and its current game state to three XML files. The planner is initiated by BOO as a subprocess and it reads in the data stored on the XML files. After planning, the planner publishes a population of good solutions to a text file. From this set of good solutions, agent BOO selects the "best" one and begins execution. At this point, agent BOO is acting as a decision maker. The agent is hard-coded by an expert of the BA RTS domain, to prefer a build-order of the form $(0, 0, x)$ unless a build-order of the form $(a, 0, y)$ satisfies the following expression:

$$a + y < x \tag{4.8}$$

where x is objective three (makespan) of a unique solution, and a and y are the objective one and objective three, respectively, of a different solution. This allows the agent to favor build-orders that do not reach the goal, but have a better overall makespan had the action string been able to encode the remaining actions required to complete the build-order plan. Recall that a fitness value of $(0, 0, x)$ describes a solution that reaches the goal with a makespan of x.

It is important to note that as the planner initializes its population it first reads in expert build-orders from a text file and then fills the rest of the population with random solutions. The only strategy to take advantage of expert build-orders is the Expansion strategy. This expert text file of build-orders is filled with expert solutions that were discovered offline by manually running the planner for various strategic goals and from previous iterations of in game plans generated by agent BOO - this entailed post processing of agent BOO's game log files. However, it would be possible to insert these solutions via the CBR system as they are written to cases as the agent determines good builds, but that technique is not utilized for experimentation.

Table 4.17: Three Balanced Annihilation Strategies [61].
Metal Extractors(MX), Solar Panels (SP), Vehicle Plant(VP), K-Bot Lab (K-Lab), Light Laser Tower (LLT), Defender Anti-Air Tower (RL).

Strategy Name	Initial Economy	Defensive Structures	Combat Units
Tank Rush	(3)MX:(3)SP:(1)VP	0	(3)Stumpy
Expansion	(2)MX,(3)SP,(1)K-lab	(8)LLT:(7)RL	(1)Flea:(4)PeeWee:(2)Rocko:(4)Jethro:(2)Hammer:(8)Warrior
Turtle	(3)MX:(4)SP:(1)VP	(9)LLT:(9)RL	(4)Flash:(3)Samson:(8)Stumpy:(3)Janus

1. **Tank Rush:** Requires building an initial economy and then producing attack waves of three stumpy tanks. The overall objective is to minimize the time between attack waves or release attack waves as quickly as possible.

2. **Expansion:** The agent is primarily focused on developing a large attack wave and base defenses while expanding its volumetric resource control as quickly as possible.

3. **Turtle:** The focus of this strategy is to deliberately build base defenses and infrastructure to support the production of units. Slowly the agent produces its first attack wave.

4.5.2 MOEA and Simulator Parameters.

Table 4.18: Balanced Annihilation BOO Planner: NSGAII Parameters

Strategy	Population Size	Num Evals	Mutation Rate	Crossover Rate	Bit Length
Tank Rush	50	6000	1/k	0.9	k=60
Expansion	50	6000	1/k	0.9	k=80
Turtle	50	6000	1/k	0.9	k=80

Table 4.19: Balanced Annihilation BOO Planner: Simulator Parameters

Metal Collect Rate	Energy Collect Rate
2.04/sec per metal extract	20/sec per solar panel

4.5.3 Experimental Objective Three Results.

The results presented in this section are derived from the best game out of five games each agent played against the Spring Engine agent NULL AI. Agent NULL AI takes no actions throughout the game session. This is to remove interruptions in build-order executions as agent LJD and agent BOO execute their strategies. Our intention is to observe which agent executes the strategies the fastest, under the assumption that an expert RTS player always executes strategies faster than a player of a lower skill level.

Figures 4.6, 4.8, 4.10 capture the performance of agent BOO and agent LJD in executing the three strategies **Tank Rush**, **Expansion**, and **Turtle**. These figures chart the build-order time-line of the agents. The build-order time-line displays the goals and ordering of goals of the various strategies as derived from agent LJD's player vector - our expert player in terms of strategy execution and compiling the CBR case set. The associated blue and red arrows with each goal relate to the timing of a build-order executed by agents LJD and BOO respectively. The images on the arrows depict the unit or building that was produced by the agent at a time given by the timestamps above each constructed unit and building. The overall time to complete a goal is captured at the head of the arrows. The

agent that reached the goal the fastest owns a yellow star next to their arrow. Once again, the time values are in-game timestamps retrieved from the player vector of each agent once the game is completed. These timestamps have not been altered. Note that not all actions for reaching a goal are displayed on the arrows. The arrows only identify the milestone actions that define a particular strategy. For the Tank Rush strategy, additional actions that are not milestones are identified with a red dashed line box around them. For the other two strategies additional actions are left off entirely. This is explained more clearly for each strategy as the figures are presented.

At the top of each build-order time-line diagram is a legend. The legend is a red and blue arrow with the respective agent names contained in them. In addition, at the head of these two arrows are the geometric DBD values the agents achieved in executing the related strategy. The agent that achieves the best geometric DBD value possesses a yellow star next to their name. If an agent produced more buildings or units than another agent while executing the same strategy a + symbol and numeric quantity to the right of a unit or building object is presented next to the agent's name in the legend.

Directly below the build-order time-line figure is a plot featuring the arithmetic and geometric DBD values achieved by the agents.

4.5.3.1 Tank Rush Strategy Analysis.

From the build-order time-line of the Tank Rush strategy displayed in Figure 4.6, it is obvious that agent BOO executes the strategy faster than agent LJD. It can be observed in Goal one that agent BOO decides to produce a vehicle plant earlier than agent LJD. This allows the vehicle plant of agent BOO to start producing the first tank of assault wave one 13 seconds earlier than agent LJD. Notice, however, in Goal two that LJD still beats BOO in developing the first tank. This is because agent BOO decides to go and produce additional infrastructure to increase the volumetric resource collection rate in Goal two to decrease production time; whereas, agent LJD is scripted to remain with the vehicle plant and assist

it in producing tanks for all attack waves. Agent BOO then returns to the vehicle plant in Goal two to assist in completing the remaining two tanks. With the increase in metal supply, agent BOO is able to complete Goal two 3 seconds faster than agent LJD. After Goal two agent BOO enters a brief planning phase. This planning phase introduces a brief delay, on average between three to 6 seconds, to agent BOO performing any actual work. As soon as the plan for Goal three is retrieved agent BOO resumes execution. Observe that agent BOO once again decides to construct additional infrastructure to increase volumetric resource supplies. By the end of Goal three, agent BOO is now 10 seconds ahead of agent LJD in overall strategy execution. Finally, in Goal four agent BOO builds more infrastructure and ends Goal four 20 seconds ahead of agent LJD with respect to overall strategy execution. As identified in the legend of the Tank Rush strategy, agent BOO produces three additional metal extractors versus agent LJD, and one more solar panel versus agent LJD.

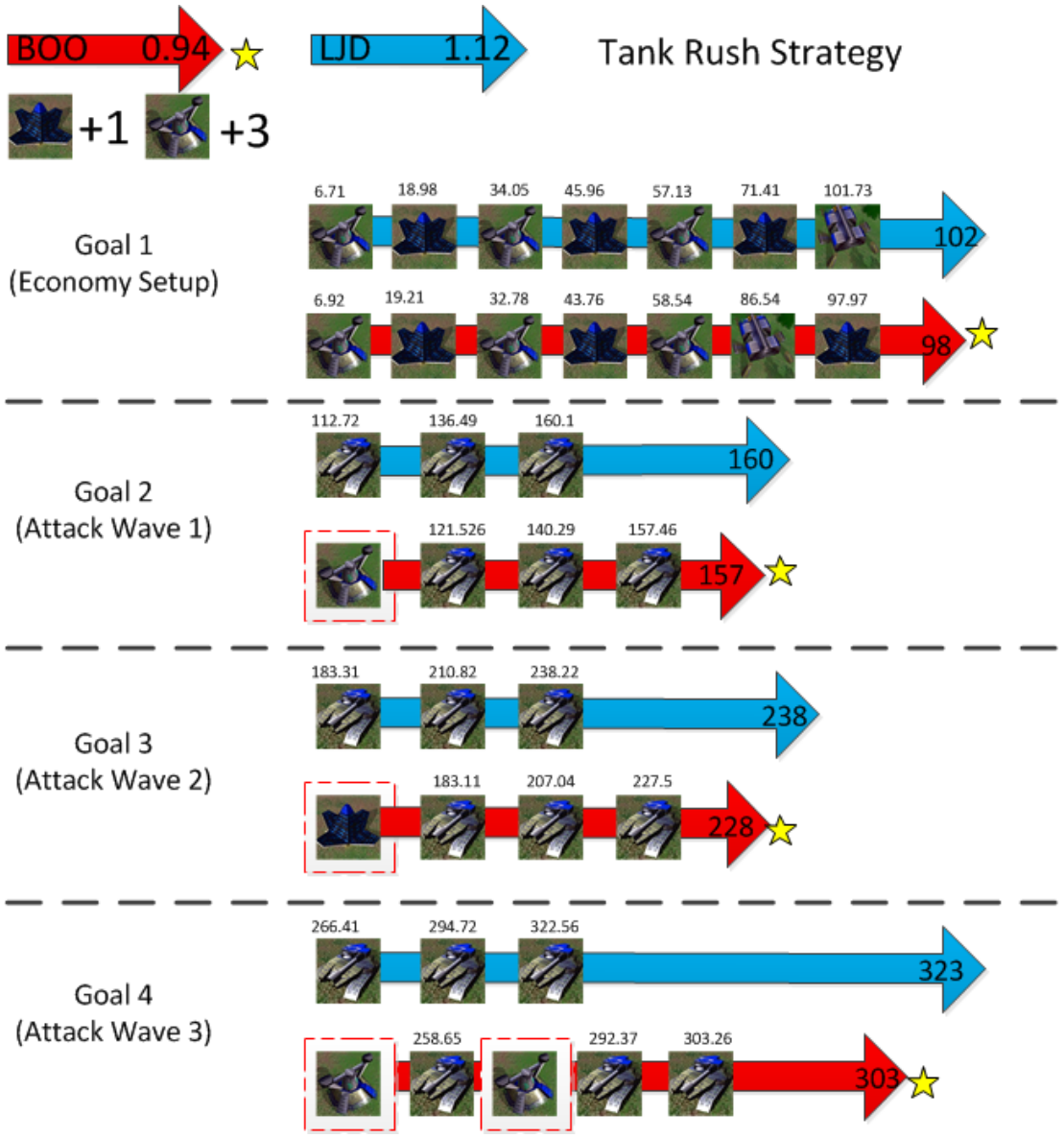

Figure 4.6: In game execution of the Tank Rush strategy. The data presented reflects the best game out of five games played against the Spring Engine agent NULL AI for agents LJD and BOO.

In Figure 4.7, the plot of the arithmetic DBD shows that agent BOO completes the overall strategy execution faster than agent LJD. In addition, the geometric DBD reveals that agent BOO shows a tighter timing consistency in reaching strategy milestones. This

is because each tank produced is considered a strategy milestone including the release of attack waves. The time required to construct the additional infrastructure was insignificant in comparison to the speed in constructing the nine tanks for the first three attack waves.

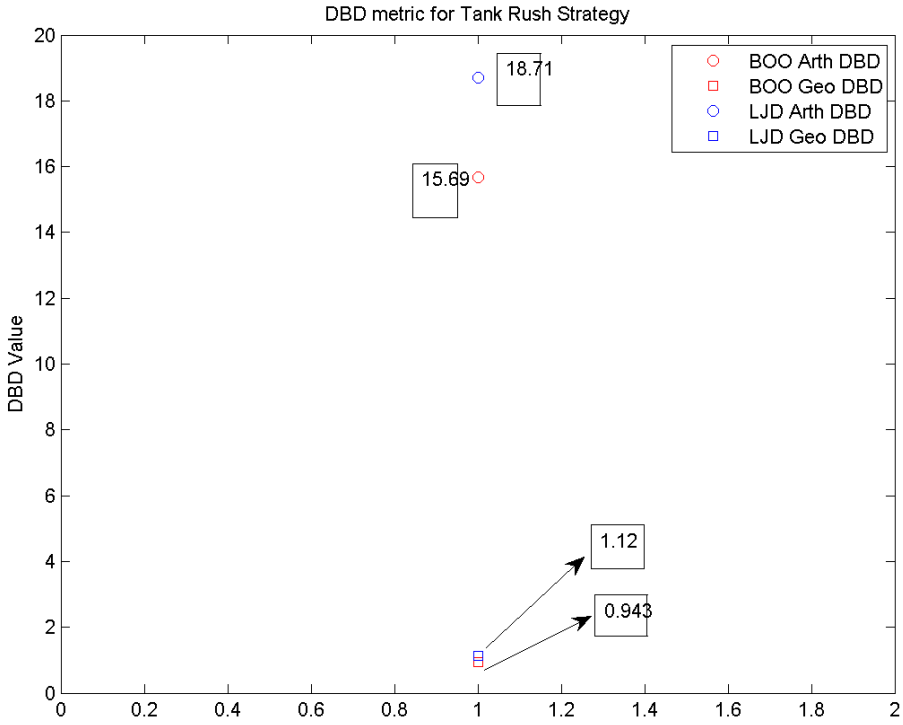

Figure 4.7: The arithmetic and geometric DBD scores of agents LJD and BOO for the Tank Rush strategy.

4.5.3.2 Expansion Strategy Analysis.

The build-order time-line Figure 4.8 of the Expansion strategy is slightly altered from the diagram of the Tank Rush strategy. This is due to the fact that the Expansion strategy is more complex than the Tank Rush strategy. By complex, we mean that the Expansion strategy requires a larger number of diverse decision-making. In the Tank Strategy the agents are only required to build Stumpy tanks and any required infrastructure to support

the production of those units. In the Expansion strategy, observed in Table 4.17, the strategy requires the production of five unique combat unit types and two unique defensive structures, while also supporting volumetric resource production requirements. Notice, in Figure 4.8 that the goals are described by unit pictures with a numerical value to their right side. This translates as every image to the left of a number must be produced in a quantity equal to that number.

Observe in Goal one the unique build-orders generated by both agents - agent BOO decides to construct the k-bot factory as soon as possible. This enabled agent BOO to start unit production 40 seconds earlier than agent LJD. Agent BOO completes Goal one 46 seconds before agent LJD. This trend continues all the way to Goal three where agent BOO releases its first attack wave 54 seconds ahead of agent LJD. Agent BOO continues to surpass agent LJD in goal completion time up until Goal six where agent LJD beats agent BOO by 9 seconds. Observe, however, that agent BOO constructs an additional air-defense tower in Goal six. In addition, agent BOO constructs an additional five solar panels versus agent LJD's additional metal extractor over the course of the entire execution of the strategy. The times shown also include the planning time in which agent BOO is not executing any actions for 3 to 6 seconds.

The DBD plot in figure 4.9 provides insight into the overall execution skill-level of the agents in executing the Expansion strategy. According to the arithmetic DBD, agent BOO executes the overall strategy faster than agent LJD; however, according to the geometric DBD, agent LJD displays a higher expert timing consistency in reaching strategy milestones. This is most likely due to the fact that agent BOO constructs six additional units/buildings (five solar panels and one additional anti-air tower) more than agent LJD over the course of the strategy execution. Also contributing to this is the planning time between build-orders. It is expected that agent BOO should execute the strategy faster than or as fast as agent LJD, which the results support.

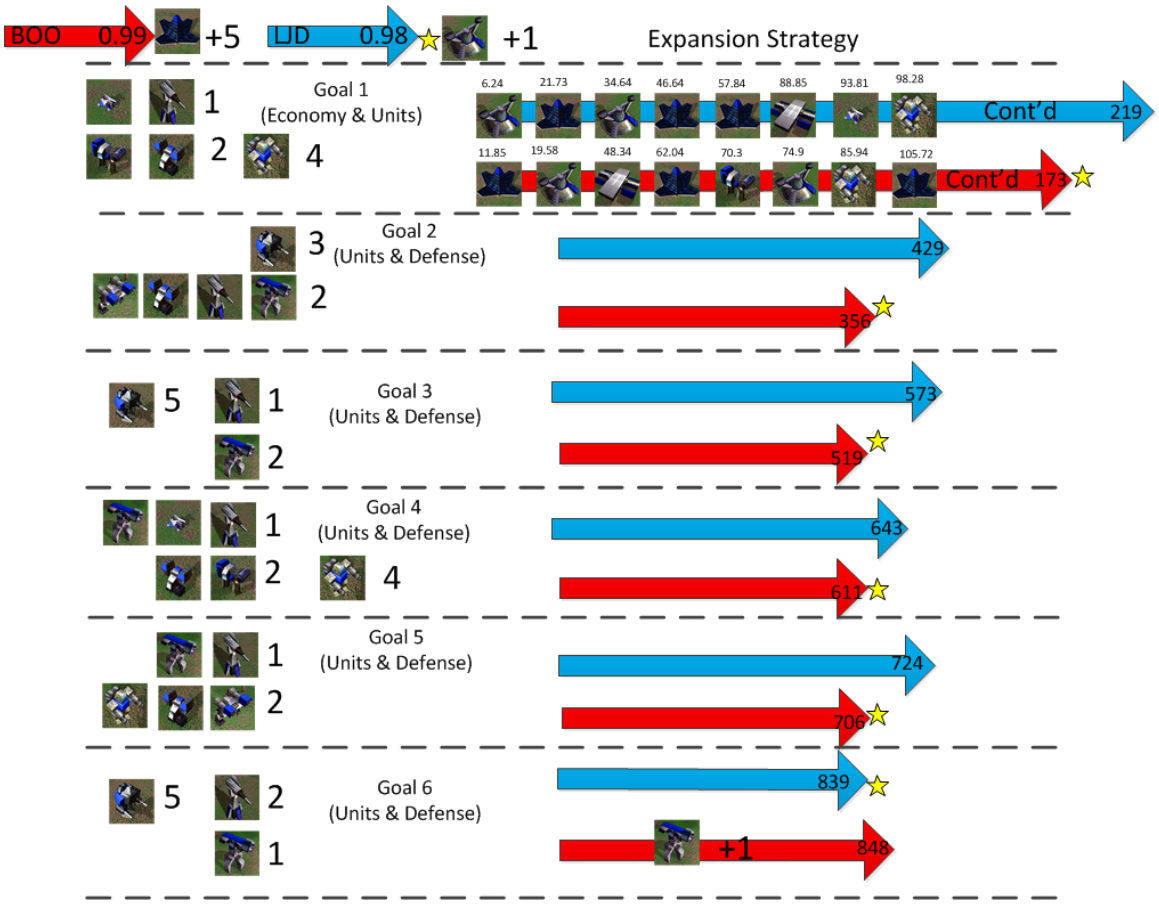

Figure 4.8: In game execution of the Expansion strategy. The data presented reflects the best game out of five games played against the Spring Engine agent NULL AI for agents LJD and BOO.

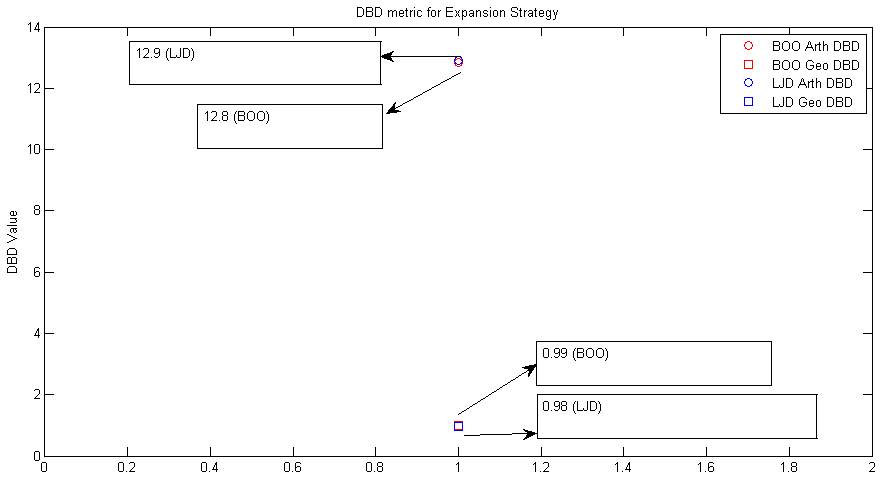

Figure 4.9: The arithmetic and geometric DBD scores of agents LJD and BOO for the Expansion strategy.

4.5.3.3 Turtle Strategy Analysis.

In executing the Turtle strategy, agent BOO once again prevails in executing the overall strategy faster than agent LJD. Observe that by the 2nd Attack Wave, captured as the last pair of arrows in the build-order time-line Figure 4.10, agent BOO has a 31 second lead on agent LJD. Agent BOO executes all goals faster than agent LJD. However, note that agent BOO also produces three additional solar panels more than agent LJD. According to arithmetic DBD values 4.11, agent BOO in fact executes the strategy faster than agent LJD, however, agent LJD once again displays a faster timing consistency between milestone strategy decisions according to geometric DBD. This is again due to the additional structures produced by agent BOO and the planning time between build-orders.

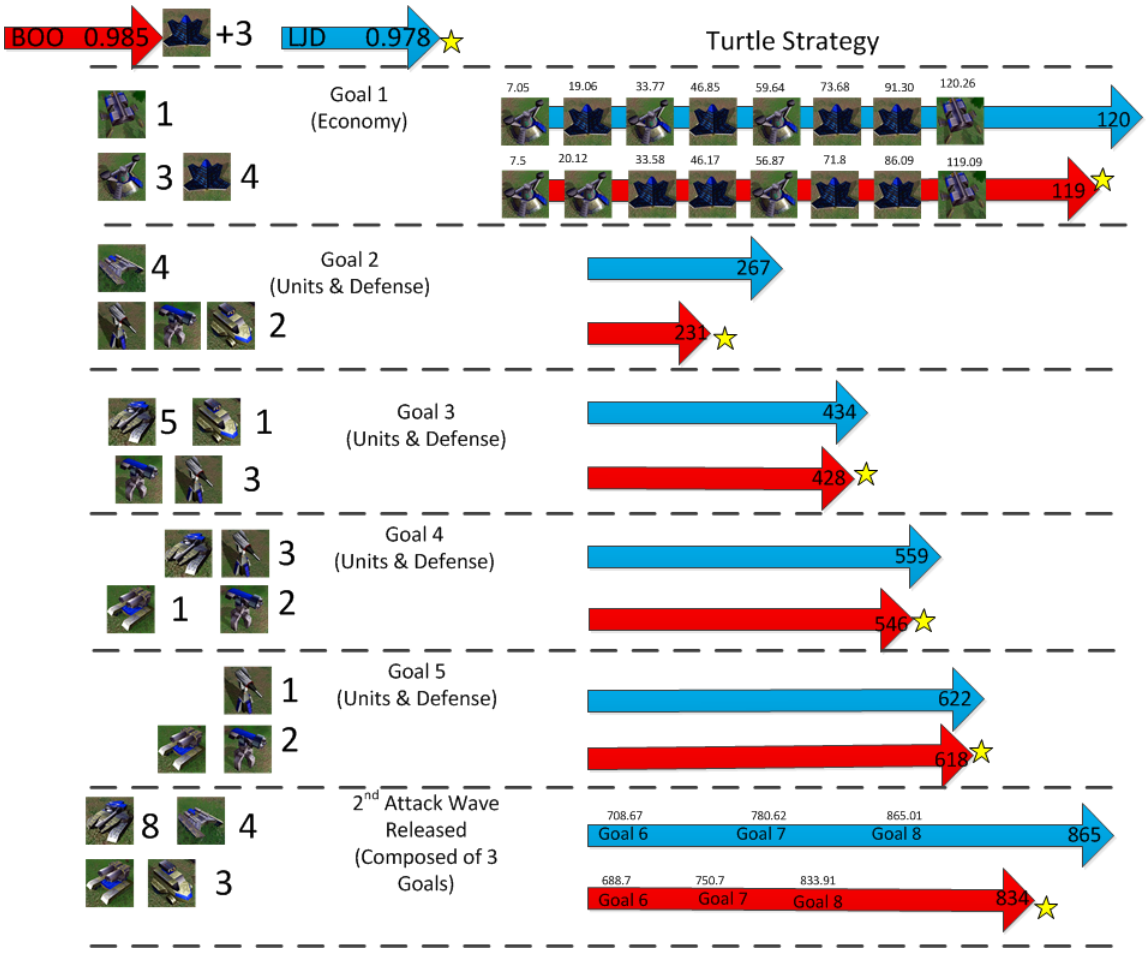

Figure 4.10: In game execution of the Turtle strategy. The data presented reflects the best game out of five games played against the Spring Engine agent NULL AI for agents LJD and BOO.

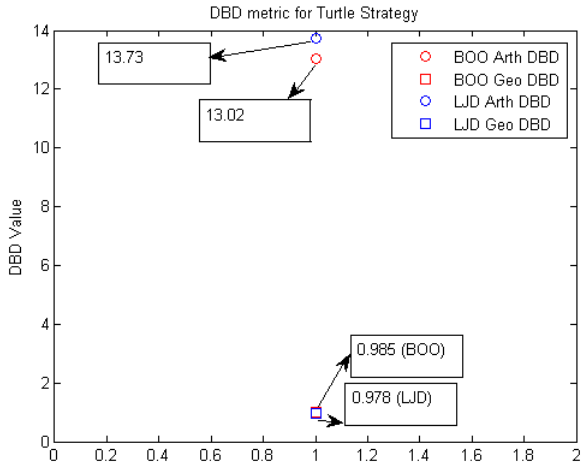

Figure 4.11: The arithmetic and geometric DBD scores of agents LJD and BOO for the Turtle strategy.

4.5.3.4 Visualizing Adaptability and Play.

As part of our third experimental objective, we observe agent BOO playing against the Spring Engine AI E323. Agent Boo executes the Turtle strategy, which provides E323 the time necessary for it to produce a first attack wave of two PeeWee (light infantry) bots to assault agent BOO's base of operations. The first attack wave of E323 destroys two volumetric resource collection buildings including a solar panel and a metal extractor. After eliminating the threat and entering a new planning cycle, the strategic planner aids agent BOO in deciding to reconstruct the lost infrastructure. The infrastructure is rebuilt in time for agent BOO to achieve the first Goal of the Turtle strategy on time with respect to agent LJD. This demonstrates the ability of the planner to plan in a dynamic game state. This demonstrates that the agent is able to rationalize and plan according to changes in its environment utilizing our planner. However, it is important to observe that if the buildings are destroyed during a planning cycle, the planner would not adapt the agents build-order or goals until the next planning cycle when game state information is provided to the planner. Agent BOO does go on to defeat agent E323, however, we do not play enough games to

provide a win/loss ratio. We only add this observation to demonstrate that agent BOO is capable of playing a full RTS campaign against a fully functional Spring Engine AI and even a human player.

With respect to achieving the expert capability of being adaptive, our agent is adaptive to changing game states, but only changes its goals after each planning cycle utilizing the CBR case selection mechanism. The selection mechanism examines the agent's current game state and then selects a case from the CBR case-set for the agent to plan toward. Recall that a case represents an expert goal to plan toward. The goals themselves are not being adjusted or adapted, rather a different goal is being selected to better match the change in the agent's game state. With the injection of expert solutions related to the case retrieved from the CBR, the expert build-orders are adapted or refitted via our MOEA to produce a new near-optimal build-order. This means that the adaptation of plans occurs not as a separate step as is done in other CBR systems, but is apart of our planning process. However, this adaptive quality is unlike other adaptive techniques such as goal driven autonomy (GDA) as discussed in [68]. Goal driven autonomy empowers an agent with the ability to adapt or modify goals by detecting discrepancies as the agent is planning.

4.5.3.5 Summary of Results.

A summary of the results presented in figures 4.6, 4.8, 4.10 is presented in table 4.20. As observed in table 4.20 agent LJD achieves a higher geometric DBD then agent BOO for the Expansion and Turtle strategies. Recall that geometric DBD measures the consistency of timing between milestone or key decisions from the strategy schema executed by a player. With this in mind, notice that for the Expansion and Turtle strategies agent BOO produces additional infrastructure as displayed in figures 4.8 and 4.10. These additional builds and the planning time between build-orders introduced greater time between milestone strategy decisions, which justifies why BOO would have a slightly higher geometric DBD, but lower arithmetic DBD than LJD. In addition, the similar

geometric DBD values signify that LJD and BOO are both executing a strategy at an Expert level, but the arithmetic DBD signifies that BOO is a degree more of an expert than LJD in terms of overall strategy execution. Agent BOO is in fact producing better build-orders than the expert agent LJD via our strategic planner.

Table 4.20: Goal Strategy Execution Timeline versus NullAI. Waves identify the release times of attack waves in seconds.

Agent Name	Strategy	Wave 1 (secs)	Wave 2	Wave 3	Wave 4	Wave 5	Arth/Geo DBD
LJD	Tank Rush	160.16	238.24	322.57	399.51	480.71	18.71/1.12
BOO	Tank Rush	**157.49**	**227.57**	**303.27**	**353.07**	**423.5**	**15.69/0.94**
LJD	Expansion	573.08	**838.89**	-	-	-	12.9/**0.98**
BOO	Expansion	**519.90**	847.96	-	-	-	**12.8**/0.99
LJD	Turtle	622.33	865.01	-	-	-	13.73/**0.978**
BOO	Turtle	**617.52**	**833.91**	-	-	-	**13.02**/0.985

4.6 Summary of Experimental Results

Chapter 4 has validated our approach to online strategic planning for RTS agents. Through the use of the MO-BOO problem, a generic RTS strategic decision-making simulator framework, a MOEA, and CBR we have successfully designed and implemented a new AI technique for implementing the strategic manager of a multi-scale agent. Our design supports online planning in a fully operational RTS game as demonstrated with agent BOO. Agent BOO is capable of planning through the first technology level of the RTS BA game, which includes 21 possible actions, at a performance level greater than or equal to an expert scripted AI. This is achieved without bounding actions or constraining the RTS decision space.

V. Conclusion and Final Remarks

In chapter one we presented our research objectives and set out to achieve the following three target objectives:

1. **Develop (design, implement and validate) an on-line capability to determine the skill level of a RTS player**

2. **Develop an online RTS multi-objective AI planning algorithm for generating human expert level RTS build-orders [22]**

3. **Integrate on-line planning tool with an agent in the Spring RTS game engine Balanced Annihilation game [3] and validate via simulation**

Through our experimental objectives outlined in chapter 4 and reprinted below, we evaluate each one of our target research objectives.

1. **Experimental objective one of this work is to validate that the mathematical model of the build-order problem we formulated is capable of planning in the RTS domain. This is demonstrated across planning goals for Starcraft: Broodwars and Balanced Annihilation.**

2. **Experimental objective two is to demonstrate the capabilities of the planning tool across varying parameter settings. This is conceptualized via a Pareto front.**

3. **Experimental objective three is to demonstrate that our strategic planning tool, utilizing the mathematical model, can be utilized as an online planning tool by a RTS agent to play through a full RTS game at or near an expert level.**

4. **Experimental objective four is to validate a method for determining player skill-level in an RTS game. This objective is achieved in connection with the first three experimental objectives.**

Experimental objective one demonstrates the ability of our MO-BOO problem formulation to effectively approximate the build-order problem for cumulative and non-cumulative economy RTS games like Starcraft and BA. In addition, experimental objective one also validates our strategic planning tool's ability to reach user defined goals via simulation and a MOEA without the use of expert data. Experimental objective one confirms that we have achieved research objective **2a**.

Experimental objective two evaluates the performance of our strategic planning design for various parameter settings for the two RTS simulators and NSGAII algorithm. It also provides a depiction of the Pareto front for the RTS MO-BOO problem. This experiment validates the ability of our planner to reach goals in both an offline and online state. It further reinforces the use of intermediate planning over singular planning as contended in [19]. This experiment supports research objectives **2a** and **2g**.

Experimental objective three validates the use of our strategic planning tool as an online planning capability for RTS multi-scale agents. It demonstrates the ability of our RTS agent BOO to perform at or exceed the skill level of an expert scripted RTS agent from which its goals and goal-ordering are derived. The significance of this is that it is possible for an RTS agent utilizing our planning tool with CBR to perform better than or equal (in terms of strategy execution) to the expert players from which the CBR case set is compiled. This experiment also validates the use of our skill-level metric DBD as a means of measuring skill-level relative to a set of known experts. Experimental objective three confirms the achievement of research objectives **1a** and **2g**.

5.1 Contributions

We provide the RTS research community with a concise mathematical model of the RTS build-order problem. The build-order optimization problem consists of three objective functions and three constraints and is derived from the efforts of a multitude of researchers with strong influences from [71], [20], [22].

In addition we provide two generic RTS simulator frameworks and Python implementations for simulating strategic decision making in RTS games including games like Starcraft I and II, Age of Empires, Wargus, Total Annihilation, and Balanced Annihilation. We also introduce the idea of cumulative and non-cumulative economy RTS games to distinguish the economical differences between games like Starcraft versus BA. In combination with the JmetalCpp framework and our XML schema files packaged with the simulators, we provide a build-order planning tool for non-cumulative economy games like Total Annihilation and BA, and for cumulative economy games like Starcraft and AOE. By modifying simple XML schema files any researcher can continue with our work in their selected RTS game including Wargus. The planners can be utilized offline or integrated into the games for online use - note that the code must be translated into C++ or Java to reduce computational costs for online use as we demonstrated for BA in experimental objective three.

Finally, we provide a unique, online multi-objective approach to strategic planning in RTS games. Our strategic planner blends ideas from behavioral AI approaches and optimization approaches. It integrates our build-order mathematical model, our simulators, a MOEA, and CBR to produce expert level or near-optimal build-orders for an agent in the Spring RTS game engine in real-time. Through experimental objectives one through four we have demonstrated the ability of our agent BOO to execute strategies at an expert level relative to Agent LJD and the BA game. Our agent effectively enters the game with no domain knowledge, but through our mathematical model and CBR is able to produce expert level build-orders that lead agent BOO to execute a strategy faster than the expert from which the goals and goal ordering are derived. In addition, bounding the search space which is done for global search algorithms like A* or DFS is not required. This reduces the complexity of tuning the design for implementation in any RTS game.

In addition to the items mentioned above, we also provide a unique and needed method to measuring skill level of RTS players. Our method is effective for measuring the ability

of a player to execute a strategy. We have effectively developed an approach for measuring strategic skill level, but not for measuring tactical skill level. This method can be utilized online during game play or offline to derive a high-level model of the players skill level with respect to strategy execution. This information can be provided to an agent in order to adjust the game experience for the human player based upon their strategic skill level - this is necessary to ensure a game is both fun and challenging for players at any skill level and for education purposes in order to keep students engaged [35]

5.2 Future Work

Future efforts by our team and future members include the following list:

1. Enhance the command set defined in the XML command schema file of BA to encompass building through all tech levels.

2. Enable Agent BOO and Agent LJD to utilize construction vehicles to their full capacity - currently they can only assist the commander in the commander's construction projects.

3. Enhance the RTS simulators with map information to eliminate the travel time variable.

4. Enhance the CBR goal selection mechanism of the Agent with a Goal Autonomy Driven approach similar to the one described in [68] to produce a more adaptive agent.

5. Parallelize our strategic planning tool to further reduce computation time required to return a near-optimal build-order.

6. Develop a robust tactical planning manager for agent BOO to enhance the ability of BOO to play at an expert level.

7. Incorporate a scouting manager to allow agent BOO to adapt to opponent decision-making or strategy

8. Dynamic content injection into RTS game for the purposes of decision-making education

9. Adjustable difficulty settings based upon player's dynamic skill to support a challenging and fun educational experience

10. Tailored feedback to player in real-time to enhance critical decision-making education

5.3 Final Remarks

We are in the process of developing a robust RTS AI framework for Air Force education and training purposes. As part of this goal we are designing and implementing a multi-scale agent as the centerpiece of our training system. With the conclusion of this work we now have a robust strategic planning manager and will continue forward to incorporate a robust tactical and scouting manager. Once we have an agent capable of playing an RTS game at or near an Expert level, we seek to utilize this expert agent in an education system to evaluate a player in real-time and introduce new content or change the state of the game for the benefit of educational objectives.

Appendix: Starcraft Simulator XML Schema Files

STARCRAFT COMMAND.XML SCHEMA FILE

```
<command>
    <action>
        <name>Build_Barracks</name>
        <duration>80</duration>
        <volumetric_dictionary>
            <volumetric key="Mineral" value="150"></volumetric>
        </volumetric_dictionary>
        <unary_list>
            <unary>Worker</unary>
        </unary_list>
        <exist_list>

        </exist_list>
        <effect_list>
            <effect key="Increment" value="Barracks_Bldg"></effect>
        </effect_list>
    </action>

    <action>
        <name>Build_CmdCenter</name>
        <duration>120</duration>
        <volumetric_dictionary>
            <volumetric key="Mineral" value="400"></volumetric>
        </volumetric_dictionary>
        <unary_list>
            <unary>Worker</unary>
        </unary_list>
        <exist_list>

        </exist_list>
        <effect_list>
            <effect key="Increment" value="Command_Center"></effect>
        </effect_list>
    </action>

    <action>
        <name>Build_Refinery</name>
        <duration>40</duration>
        <volumetric_dictionary>
            <volumetric key="Mineral" value="100"></volumetric>
        </volumetric_dictionary>
        <unary_list>
            <unary>Worker</unary>
        </unary_list>
        <exist_list>

        </exist_list>
        <effect_list>
            <effect key="Increment" value="Refinery_Bldg"></effect>
        </effect_list>
    </action>
```

```
    <action>
        <name>Build_SupplyDepot</name>
        <duration>40</duration>
        <volumetric_dictionary>
            <volumetric key="Mineral" value="100"></volumetric>
        </volumetric_dictionary>
        <unary_list>
            <unary>Worker</unary>
        </unary_list>
        <exist_list>

        </exist_list>
        <effect_list>
            <effect key="Accumulate_Fixed" value="Supply"></effect>
        </effect_list>
    </action>

    <action>
        <name>Build_Worker</name>
        <duration>20</duration>
        <volumetric_dictionary>
            <volumetric key="Mineral" value="50"></volumetric>
            <volumetric key="Supply" value="1"></volumetric>
        </volumetric_dictionary>
        <unary_list>
            <unary>Command_Center</unary>
        </unary_list>
        <exist_list>

        </exist_list>
        <effect_list>
            <effect key="Increment" value="Worker"></effect>
        </effect_list>
    </action>

    <action>
        <name>Build_Marine</name>
        <duration>20</duration>
        <volumetric_dictionary>
            <volumetric key="Mineral" value="50"></volumetric>
            <volumetric key="Supply" value="1"></volumetric>
        </volumetric_dictionary>
        <unary_list>
            <unary>Barracks_Bldg</unary>
        </unary_list>
        <exist_list>

        </exist_list>
        <effect_list>
            <effect key="Unit_Increment" value="Marine"></effect>
        </effect_list>
    </action>
```

```
    <action>
        <name>Mineral</name>
        <duration>0</duration>
        <volumetric_dictionary>
        </volumetric_dictionary>
        <unary_list>
            <unary>Worker</unary>
        </unary_list>
        <exist_list>
            <exist>Command_Center</exist>
        </exist_list>
        <effect_list>
            <effect key="Hold" value="Worker"></effect>
            <effect key="Accumulate" value="Mineral"></effect>
        </effect_list>
    </action>

    <action>
        <name>Gas</name>
        <duration>0</duration>
        <volumetric_dictionary>
        </volumetric_dictionary>
        <unary_list>
            <unary>Worker</unary>
        </unary_list>
        <exist_list>
            <exist>Refinery_Bldg</exist>
        </exist_list>
        <effect_list>
            <effect key="Hold" value="Worker"></effect>
            <effect key="Accumulate" value="Gas"></effect>
        </effect_list>
    </action>

</command>
```

STARCRAFT UNARY_RESOURCE.XML SCHEMA FILE

```
<unary_resource>
    <unary name="Command_Center" exist="1"></unary>
    <unary name="Barracks_Bldg" exist="0"></unary>
    <unary name="Refinery_Bldg" exist="0"></unary>
    <unary name="Worker" exist="5"></unary>
</unary_resource>
```

STARCRAFT VOL_RESOURCE.XML SCHEMA FILE

```
<vol_resource>
    <volumetric name="Mineral" rate="1.35" init_amt="100"></volumetric>
    <volumetric name="Gas" rate="2.1" init_amt="0"></volumetric>
    <volumetric name="Supply" rate="-10" init_amt="5"></volumetric>

</vol_resource>
```

STARCRAFT COMBAT_UNIT.XML SCHEMA FILE

```
<combat_unit>
    <unit name="Marine" init_amt="0"></unit>

</combat_unit>
```

STARCRAFT UNARY_GOAL.XML SCHEMA FILE

```
<unary_resource>
    <unary name="Command_Center" exist="1" minCost="400" gasCost="0" suppCost="0" durAmt ="120"
    ></unary>
    <unary name="Barracks_Bldg" exist="0" minCost="150" gasCost="0"  suppCost="0" durAmt = "80"
    ></unary>
    <unary name="Refinery_Bldg" exist="0" minCost="100" gasCost="0"  suppCost="0" durAmt ="40"
    ></unary>
    <unary name="Worker" exist="5" minCost = "50" gasCost ="0" suppCost="1" durAmt="20"></unary>

</unary_resource>
```

STARCRAFT VOL_GOAL.XML SCHEMA FILE

```
<vol_resource>
    <volumetric name="Mineral" rate="1.0" init_amt="200"></volumetric>
    <volumetric name="Gas" rate="1.0" init_amt="200"></volumetric>
    <volumetric name="Supply" rate="-10" init_amt="0"></volumetric>

</vol_resource>
```

STARCRAFT UNIT_GOAL.XML SCHEMA FILE

```
<combat_unit>
    <unit name="Marine" exist="7.0" minCost="50" gasCost="0" suppCost="1" durAmt="20"></unit>
</combat_unit>
```

Bibliography

[1] "Agent Planning Code". URL https://code.google.com/p/online-planning-rts-agents/.

[2] "Air Force Capabilities". URL http://www.airforce.com/learn-about/our-mission/.

[3] "Balanced Annihilation". URL http://www.balancedannihilation.org/.

[4] "Squadron Officer College". URL http://www.au.af.mil/au/.

[5] "Theater Airpower Visualization". URL http://www.strategypage.com/military_photos/military_photos_20069291717.aspx.

[6] "WEKA". URL http://www.cs.waikato.ac.nz/ml/weka/.

[7] Aha, DW, Matthew Molineaux, and Marc Ponsen. "Learning to win: Case-based plan selection in a real-time strategy game". *Sixth International Conference on Case-Based Reasoning*, 5–20. Springer, 2005. URL http://link.springer.com/chapter/10.1007/11536406_4.

[8] Allen, James F. "Maintaining knowledge about temporal intervals". *Communications of the ACM*, 26(11):832–843, November 1983. ISSN 00010782. URL http://portal.acm.org/citation.cfm?doid=182.358434.

[9] Avontuur, Tetske. *Modeling player skill in Starcraft II*. Master's thesis, 2012.

[10] Balla, RK and Alan Fern. "UCT for Tactical Assault Planning in Real-Time Strategy Games." *International Joint Conferences on Artificial Intelligence*, 40–45, 2009. URL http://www.aaai.org/ocs/index.php/IJCAI/IJCAI-09/paper/viewPDFInterstitial/632/587.

[11] Ballinger, C and S Louis. "Finding Robust Strategies to Defeat Specific Opponents Using Case-Injected Coevolution". *IEEE Conference on Computational Intelligence and Games*. 2013. ISBN 9781467353113.

[12] Baptiste, Philippe, Claude L E Pape, and S A Ilog. "Disjunctive Constraints for Manufacturing Scheduling: Principles and Extensions". *International Journal of Computer Integrated Manufacturing*, 9(4):306–310, 1996. URL http://www.tandfonline.com/doi/abs/10.1080/095119296131616#.UllpGVCkrpo.

[13] Baptiste, Philippe and Claude Le Pape. "Edge-Finding Constraint Propagation Algorithms for Disjunctive and Cumulative Scheduling". *Scheduling, Proceedings 15th Workshop of the U.K. Planning Special Interest Group*. 1996.

[14] Branquinho, A, CR Lopes, and TF Naves. "Using search and learning for production of resources in rts games". *23rd IEEE International Conference on Tools with Artificial Intelligence*, 2011. URL http://world-comp.org/p2011/ICA4302.pdf.

[15] Browne, Cameron, Edward Powley, Daniel Whitehouse, Simon Lucas, Senior Member, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. "A Survey of Monte Carlo Tree Search Methods". *IEEE Conference on Computational Intelligence and AI in Games*, 4(1):1–49, 2012.

[16] Buro, Michael. "Real-Time Strategy Games: A New AI Research Challenge". *18th International Joint Conference on Artificial Intelligence*, 1534–1535. International Joint Conferences on Artificial Intelligence, 2003.

[17] Bylander, Tom. "Complexity Results for Planning". *Proc. IJCAI-91*, 274–279. 1991.

[18] Bylander, Tom. "Concurrent Action Selection with Shared Fluents". *Proc. AAAI Vancouver, Canada*, 274–279. 2007.

[19] Chan, Hei, Alan Fern, and Soumya Ray. "Extending online planning for resource production in real-time strategy games with search". *Workshop on Planning in Games, ICAPS*, 2007. URL http://vorlon.case.edu/~sray/papers/meals-workshop.icaps07.pdf.

[20] Chan, Hei, Alan Fern, Soumya Ray, Nick Wilson, and Chris Ventura. "Online Planning for Resource Production in Real-Time Strategy Games". *In Proc. of the In. Conference on Automated Planning and Scheduling*, 2007.

[21] Chung, Michael, Michael Buro, and Jonathan Schaeffer. "Monte Carlo Planning in RTS Games". *IEEE Symposium on Computational Intelligence and Games*, 117–124. 2005.

[22] Churchill, David and Michael Buro. "Build Order Optimization in StarCraft." *7th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 14–19. 2011. URL http://www.aaai.org/ocs/index.php/AIIDE/AIIDE11/paper/viewPDFInterstitial/4078/4407.

[23] Churchill, David and Michael Buro. "Incorporating Search Algorithms into RTS Game Agents". *In Proceedings of the AIIDE Conference*, 14–19. 2012. URL http://www.aaai.org/ocs/index.php/AIIDE/AIIDE12/paper/viewPDFInterstitial/5521/5733.

[24] Coello Coello, Carlos, Gary Lamont, and David Veldhuizen. *Evolutionary Algorithms for Solving Multi-objective Problems*. Springer, 2nd edition, 2007.

[25] Cojocar, William J. "Adaptive Leadership in the Military Decision Making Process." *Military Review*, 6:29–34, November-December 2011.

[26] Cornelissens, Trijntje and Helmut Simonis. "Modelling Producer / Consumer Constraints". *Workshop on Constraint Languages and Their use in Problem Modelling*. Ithaca, New York, 1995.

[27] Coy, Josh M C and Michael Mateas. "An Integrated Agent for Playing Real-Time Strategy Games". *in Proceedings of the AAAI Conf. on Artificial Intelligence*, 1313–1318. 2008.

[28] Cunha, Renato and Luiz Chaimowicz. "An Artificial Intelligence System to Help the Player of Real-Time Strategy Games". *Proceedings of the 2010 Brazilian Symposium on Games and Digital Entertainment*, SBGAMES '10, 71–81. IEEE Computer Society, Washington, DC, USA, 2010. ISBN 978-0-7695-4359-8. URL http://dx.doi.org/10.1109/SBGAMES.2010.23.

[29] Deb, Kalyanmoy. "Multi-objective Genetic Algorithms : Problem Difficulties and Construction of Test Problems". *IEEE Transactions on Evolutionary Computation*, 7(1995):205–230, 1999.

[30] Deb, Kalyanmoy, Associate Member, Amrit Pratap, Sameer Agarwal, and T Meyarivan. "A Fast and Elitist Multiobjective Genetic Algorithm :". *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[31] Echegoyen, Carlos, Qingfu Zhang, and San Sebastian. *Analyzing limits of effectiveness in different implementations of estimation of distribution algorithms*. Technical Report January, University of the Basque Country, 2011.

[32] Farkas, Bart. *Starcraft Strategy Guide*. Prima, Rocklin, California, 1998.

[33] Gmeiner, Bjorn, Donnert Gerald, and Kostler Harald. "Optimizing Opening Strategies in a Real-time Strategy Game by a Multi-objective Genetic Algorithm". *Research and Development in Intelligent Systems*, XXIX, 2012.

[34] Gmeiner, Bjorn, Donnert Gerald, and Kostler Harald. "Multi-Objective Assessment of Pre-Optimized Build Orders Exemplified for Starcraft 2". *Computational Intelligence in Games IEEE*, 2013.

[35] Hays, Robert T. *The Science of Learning: a Systems Theory Perspective*. BrownWalker, Boca Raton,FL, 2006.

[36] Khan, Nazan, Nazan Khan, David E. Goldberg, David E. Goldberg, Martin Pelikan, and Martin Pelikan. *Multi-Objective Bayesian Optimization Algorithm*. Technical report, in Proceedings of the Genetic and Evolutionary Computation Conference, 2002.

[37] Kovarsky, Alex and Michael Buro. "A First Look at Build-Order Optimization in Real-Time Strategy Games". *In Proceedings of the GameOn Conference*, 18–22. 2006. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.5338&rep=rep1&type=pdf.

[38] Langley, Pat and Dongkyu Choi. "A unified cognitive architecture for physical agents". *Proceedings of the National Conference on Artificial Intelligence*, 1469–1474. 2006. URL http://www.aaai.org/Papers/AAAI/2006/AAAI06-231.pdf.

[39] Lee, Jaeyong, Bonjung Koo, and Kyungwhan Oh. "State space optimization using plan recognition and reinforcement learning on RTS game". *7th WSEAS Int. Conf. on Aritificial Intelligence, Knowledge, Engineering and Data Bases*, 165–169. University of Cambridge, UK, 2008. ISBN 9789606766411. URL http://www.wseas.us/e-library/conferences/2008/uk/AIKED/AIKED-22.pdf.

[40] Lehman, Jill Fain, John Laird, and Paul Rosenbloom. "A gentle introduction to Soar, an architecture for human cognition". *In S. Sternberg and D. Scarborough (Eds), Invitation to Cognitive Science*. MIT Press, 1996.

[41] Mariano, CE and Eduardo Morales. "MOAQ: An Ant-Q algorithm for multiple objective optimization problems". *Proceedings of the Genetic and Evolutionary and Computation Conference*, 1999.

[42] Miles, Chris. *Case-Injected Genetic Algorithms in Computer Strategy Games*. Master's thesis, University of Nevada, 2007.

[43] Nebro, A.J., J.J. Durillo, Carlos Coello Coello, F Luna, and E. Alba. "A Study of Convergence Speed in Multi-Objective Metaheuristics". *Parallel Problem Solving from Nature*, 5199:763–772, 2008.

[44] Nebro, Antonio and Juan Durillo. "jMetal 4.3 User Manual", 2013. URL http://jmetal.sourceforge.net/.

[45] Ontanon, Santiago, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. "Case-Based Planning and Execution for Real-Time Strategy Games". *Case-Based Reasoning Research and Development*, 167–178. Springer Berlin Heidelberg, 2007. URL http://link.springer.com/chapter/10.1007/978-3-540-74141-1_12.

[46] Pérez, AU. *Multi-Reactive Planning for Real-Time Strategy Games*. Dissertation, Universitat Autònoma de Barcelona, 2011. URL http://users.soe.ucsc.edu/~bweber/pubs/Slides.pdf.

[47] Platt, John C. *Sequential Minimal Optimization : A Fast Algorithm for Training Support Vector Machines*. Technical report, Microsoft Research, 1998.

[48] Ram, Ashwin, Andrew Trusty, and Santiago Onta. "Stochastic Plan Optimization in Real-Time Strategy Games Real-Time Strategy Games". *Artificial Intelligence and Interactive Digital Entertainment*, Ccl, 1–6. 2008. URL http://www.cc.gatech.edu/faculty/ashwin/papers/er-08-09.pdf.

[49] Rule, Jeffrey N. *A Symbiotic Relationship: The OODA Loop, Intuition, and Strategic Thought.* U.S. Army War College, 2013.

[50] Russell, S. and P. Norvig. *Artificial Intelligence: A Modern Approach*. 3rd edition, 2009.

[51] Russell, Stuart and Peter Norvig. “Section 13.3 Quantifying Uncertainty”. *Artificial Intelligence A Modern Approach*. Pearson, 3rd edition.

[52] Sailer, Frantisek, Michael Buro, and Marc Lanctot. “Adversarial Planning Through Strategy Simulation”. *2007 IEEE Symposium on Computational Intelligence and Games*, 80–87, 2007. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4219027.

[53] Schadd, Frederik, Sander Bakkes, and Pieter Spronck. “Opponent Modeling in Real-Time Strategy Games”. *8th Int. Conf. Intell. Games Simul*, 61–68. 2007.

[54] Sharma, Manu, MP Holmes, and JC Santamaría. “Transfer Learning in Real-Time Strategy Games Using Hybrid CBR/RL.” *International Join Conference on Artificial Intelligence*, 1041–1046, 2007. URL http://www.aaai.org/Papers/IJCAI/2007/IJCAI07-168.pdf.

[55] Shmoys, David and David Williamson. *The Design of Approximation Algorithms*. Cambridge, 2011.

[56] Sotomayor, Teresita. *Evaluating Tactical Combat Casualty Care Training Treatments Effects On Combat Medic Trainees In Light of Select Human Descriptive Characteristics*. Dissertation, University of Central Florida, 2008. URL https://docs.google.com/viewer?url=http://etd.fcla.edu/CF/CFE0002396/Sotomayor_Teresita_M_200812_PhD.pdf.

[57] Sutton, RS and AG Barto. *Introduction to Reinforcement Learning*. MIT Press, 1 edition, 1998. ISBN 0262193981. URL http://dl.acm.org/citation.cfm?id=551283.

[58] Synnaeve, Gabriel and Pierre Bessi. “A Bayesian Model for Plan Recognition in RTS Games applied to StarCraft”. *7th Artificial Intelligence and Interactive Digital Entertainment International Conference*. 2011.

[59] Talbi, El-Ghazali. *Metaheuristics From Design to Implementation*. Wiley Publishing, 2009.

[60] Team, The Wargus. “Wargus For the Stratagus RTS Game Engine”. URL http://wargus.sourceforge.net/index.shtml.

[61] Trapani, LJ Di. *A Real-time Strategy Agent Framework and Strategy Classifier for Computer Generated Forces*. Master’s thesis, 2012. URL http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA563243.

[62] Wang, Weijia and Michele Sebag. “Multi-objective Monte-Carlo Tree Search”. *Asian Conference on Machine Learning*, 507–522. 2012.

[63] Weber, B and Michael Mateas. “Using Data Mining to Model Player Experience”. *Evaluating Player Experience in Games*. Bordeaux, France, 2011. ISBN 9781450302678.

[64] Weber, Ben. "StarCraft Data Mining", 2009. URL http://eis.ucsc.edu/StarCraft_Data_Mining.

[65] Weber, Ben and Michael Mateas. "Case-Based Reasoning for Build Order in Real-Time Strategy Games". *Artificial Intelligence and Interactive Digital Entertainment*, 1–6. 2009. URL https://games.soe.ucsc.edu/sites/default/files/bweber_aiide_09.pdf.

[66] Weber, Ben G. *Integrating Learning In A Multi-Scale Agent*. Dissertation, University of California, Santa Cruz, 2012.

[67] Weber, Ben G. and Michael Mateas. "A data mining approach to strategy prediction". *2009 IEEE Symposium on Computational Intelligence and Games*, 140–147, September 2009. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5286483.

[68] Weber, Ben G, Michael Mateas, and Arnav Jhala. "Learning from Demonstration for Goal-Driven Autonomy". *26th AAAI Conference on Artificial Intelligence*. 2012.

[69] Weber, Ben G., Peter Mawhorter, Michael Mateas, and Arnav Jhala. "Reactive planning idioms for multi-scale game AI". *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, 115–122, August 2010. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5593363.

[70] Weber, BG and Michael Mateas. "Conceptual neighborhoods for retrieval in case-based reasoning". *ICCBR 09: Proceedings of the 8th International Conference on Case-Based Reasoning*. Springer-Verlag, 2009. URL http://link.springer.com/chapter/10.1007/978-3-642-02998-1_25.

[71] Wei, LZ and LW Sun. "Build Order Optimisation For Real-time Strategy Game", 2009. URL http://www.nus.edu.sg/nurop/2009/SoC/nurop-LimZhanWei.pdf.

[72] Weijers, Stefan. "Real-Time Strategy High-level Planning", 2010. URL https://www.inter-actief.utwente.nl/studiereis/pixel/files/indepth/StefanWeijers.pdf.

[73] Wintermute, Sam, Joseph Xu, and John E Laird. "SORTS : A Human-Level Approach to Real-Time Strategy AI". *Proceedings of the Third Artificial Intelligence and Interactive Digital Entertainment Conference*, 55–60. 2007. URL http://web.eecs.umich.edu/~soar/sitemaker/docs/pubs/AIIDE07-SORTS.pdf.

[74] Wolpert, David H and William G Macready. "No Free Lunch Theorems for Optimization". *IEEE Trans. Evol. Comput.*, 1(1):67–82, 1997.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
27–03–2014	Master's Thesis	Oct 2013–Mar 2014

4. TITLE AND SUBTITLE

Online Build-Order Optimization for Real-Time Strategy Agents Using Multi-Objective Evolutionary Algorithms

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Blackford, Jason M., Captain, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-14-M-13

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Intentionally Left Blank

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This investigation introduces a novel approach for online build-order optimization in real-time strategy (RTS) games. The goal of our research is to develop an artificial intelligence (AI) RTS planning agent for military critical decision-making education with the ability to perform at an expert human level, as well as to assess a players critical decision-making ability or skill-level. Build-order optimization is modeled as a multi-objective problem (MOP), and solutions are generated utilizing a multi-objective evolutionary algorithm (MOEA) that provides a set of good build-orders to a RTS planning agent. We define three research objectives: (1) Design, implement and validate a capability to determine the skill-level of a RTS player. (2) Design, implement and validate a strategic planning tool that produces near expert level build-orders which are an ordered sequence of actions a player can issue to achieve a goal, and (3) Integrate the strategic planning tool into our existing RTS agent framework and an RTS game engine. The skill-level metric we selected provides an original and needed method of evaluating a RTS players skill-level during game play. This metric is a high-level description of how quickly a player executes a strategy versus known players executing the same strategy. Our strategic planning tool combines a game simulator and an MOEA to produce a set of diverse and good build-orders for an RTS agent. Through the integration of case-base reasoning (CBR), planning goals are derived and expert build-orders are injected into a MOEA population. The MOEA then produces a diverse and approximate Pareto front that is integrated into our AI RTS agent framework. Thus, the planning tool provides an innovative online approach for strategic planning in RTS games. Experimentation via the Spring Engine Balanced Annihilation game reveals that the strategic planner is able to discover build-orders that are better than an expert scripted agent and thus achieve faster strategy execution times.

15. SUBJECT TERMS

Add four or five key words/phrases for indexing

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Gary B. Lamont (ENG)
U	U	U	UU	158	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636 x4718 gary.lamont@afit.edu